N° d'ordre

THESE

présentée à

L'UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR DE TROISIEME CYCLE

"Informatique"

par

HENRI SAVARY

—o—

OUTILS DE MISE AU POINT POUR LANGAGES DE HAUT NIVEAU: ASSOCIATION DE MODULES ET CONTROLE DE L'EXECUTION

—o—

Thèse soutenue le 15 septembre 1973 devant la commission d'examen

Président Monsieur N. GASTINEL

Examinateurs Messieurs M. GRIFFITHS
L. BOLLIET
M. BERTHAUD

Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41544319_0047

N° d'ordre

THESE

présentée à

L'UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR DE TROISIEME CYCLE

"Informatique"

par

HENRI SAVARY

—— o ——

OUTILS DE MISE AU POINT POUR LANGAGES DE HAUT NIVEAU: ASSOCIATION DE MODULES ET CONTROLE DE L'EXECUTION

—— o ——

Thèse soutenue le 15 septembre 1973 devant la commission d'examen

Président Monsieur N. GASTINEL

Examinateurs Messieurs M. GRIFFITHS

L. BOLLIET

M. BERTHAUD

UNIV.
LEIDEN
BIBL.

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

LISTE DES PROFESSEURS

Président : Monsieur Michel SOUTIF
Vice-Président : Monsieur Gabriel CAU

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Georges	Clinique des maladies infectieuses
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BENOIT Jean	Radioélectricité
	BERNARD Alain	Mathématiques Pures
	BESSON Jean	Electrochimie
	BEZES Henri	Chirurgie générale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BONNIER Etienne	Electrochimie Electrométallurgie
	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	BRAVARD Yves	Géographie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	CABANAC Jean	Pathologie chirurgicale
	CABANEL Jean	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et Toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Oto-Rhino-Laryngologie
	CHATEAU Robert	Thérapeutique
	CHENE Marcel	Chimie papetière
	COEUR André	Pharmacie chimique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie Pathologique
	CRAYA Antoine	Mécanique

Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphiligraphie
	FAU René	Clinique neuro-psychiatrique
	FELICI Noël	Electrostatique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures
	GASTINEL Noël	Analyse numérique
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques Pures
	GIRAUD Pierre	Géologie
	KLEIN Joseph	Mathématiques Pures
Mme	KOFLER Lucie	Botanique et Physilogie végétale
MM.	KOSZUL Jean-Louis	Mathématiques Pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques Pures
MM.	MALGRANGE Bernard	Mathématiques Pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Seméiologie médicale
	MASSEPORT Jean	Géographie
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et Pétrographie
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAUTHENET René	Electrotechnique
	PAYAN Jean-Jacques	Mathématiques Pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET René	Servomécanismes
	PILLET Emile	Physique industrielle
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REULOS René	Physique industrielle
	RINALDI Renaud	Physique
	ROGET Jean	Clinique de pédiatrie et de puériculture
	SANTON Lucien	Mécanique
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SILBERT Robert	Mécanique des fluides
	SOUTIF Michel	Physique générale

MM.	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLAND François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
M.	VERAIN André	Physique
Mme	VEYRET Germaine	Géographie
MM.	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	BULLEMER Bernhard	Physique
	HANO JUN-ICHI	Mathématiques Pures
	STEPHENS Michaël	Mathématiques appliquées

PROFESSEURS SANS CHAIRE

MM.	BEAUDOING André	Pédiatrie
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BONNETAIN Lucien	Chimie minérale
Mme	BONNIER Jane	Chimie générale
MM.	CARLIER Georges	Biologie végétale
	COHEN Joseph	Electrotechnique
	COUMES André	Radioélectricité
	DEPASSEL Roger	Mécanique des fluides
	DEPORTES Charles	Chimie minérale
	GAUTHIER Yves	Sciences biologiques
	GAVEND Michel	Pharmacologie
	GERMAIN Jean-Pierre	Mécanique
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	HACQUES Gérard	Calcul numérique
	JANIN Bernard	Géographie
Mme	KAHANE Josette	Physique
MM.	MULLER Jean-Michel	Thérapeutique
	PERRIAUX Jean-Jacques	Géologie et Minéralogie
	POULOUJADOFF Michel	Electrotechnique
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie générale
	ROBERT André	Chimie papetière
	DE ROUGEMONT Jacques	Neurochirurgie
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIBILLE Robert	Construction mécanique
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

Mle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBLARD Pierre	Dermatologie
	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Yves	Chimie
	BEGUIN Claude	Chimie organique
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
	BILLET Jean	Géographie
	BLIMAN Samuel	Electronique (EIE)
	BLOCH Daniel	Electrotechnique
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BOUCHET Yves	Anatomie
	BOUVARD Maurice	Mécanique des fluides
	BRODEAU François	Mathématiques (IUT B)
	BRUGEL Lucien	Energétique
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHIAVERINA Jean	Biologie appliquée (EFP)
	CHIBON Pierre	Biologie animale
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	DURAND Francis	Métallurgie
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques appliquées
	GROULADE Joseph	Biochimie médicale
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Médecine préventive
	IDELMAN Simon	Physiologie animale
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	JOLY Jean-René	Mathématiques Pures
	JOUBERT Jean-Claude	Physique du solide
	JULLIEN Pierre	Mathématiques Pures
	KAHANE André	Physique générale
	KUHN Gérard	Physique
	LACOUME Jean-Louis	Physique
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LANCIA Roland	Physique atomique
	LE JUNTER Noël	Electronique
	LEROY Philippe	Mathématiques
	LOISEAUX Jean-Marie	Physique nucléaire
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LUU DUC Cuong	Chimie organique
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et Médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)

MM.	MAYNARD Roger	Physique du solide
	MICHOULIER Jean	Physique (IUT A)
	MICOUD Max	Maladies infectieuses
	MOREAU René	Hydraulique (INP)
	NEGRE Robert	Mécanique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PELMONT Jean	Physiologie animale
	PERRET Jean	Neurologie
	PERRIN Louis	Pathologie expérimentale
	PFISTER Jean-Claude	Physique du solide
	PHELIP Xavier	Rhumatologie
Mle	RIERY Yvette	Biologie animale
MM.	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RENAUD Maurice	Chimie
	RICHARD Lucien	Botanique
Mme	RINAUDO Marquerite	Chimie macromoléculaire
MM.	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean-Claude	Chimie générale
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VAN CUTSEM Bernard	Mathématiques appliquées
	VEILLON Gérard	Mathématiques appliquées (INP)
	VIALON Pierre	Géologie
	VOOG Robert	Médecine interne
	VROUSSOS Constantin	Radiologie
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES ASSOCIES

MM.	BOUDOURIS Georges	Radioélectricité
	CHEEKE John	Thermodynamique
	GOLDSCHMIDT Hubert	Mathématiques
	SIDNEY STUARD	Mathématiques Pures
	YACOUD Mahmoud	Médecine légale

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

Mme	BERIEL Hélène	Physilogie
Mme	RENAUDET Jacqueline	Microbiologie

Fait le 30 mai 1972.

Je tiens à exprimer ma reconnaissance à :

Monsieur le Professeur Noël GASTINEL, Directeur du Laboratoire de Calcul, qui m'a fait l'honneur de présider le jury de cette thèse.

Monsieur Michaël GRIFFITHS, Maître de Conférences à l'Université Scientifique et Médicale de Grenoble, qui m'a conseillé et guidé dans mes travaux et dont j'ai apprécié l'entrain communicatif.

Monsieur Louis BOLLIET, Professeur à l'Institut Universitaire de Technologie de Grenoble, qui s'est intéressé à ce travail et a accepté de le juger.

L'équipe du Centre Scientifique IBM de Grenoble, qui m'a accueilli et en particulier :

- *Monsieur M. BERTHAUD, qui a suivi mes travaux avec sympathie.*
- *Monsieur D. CLAUZEL avec lequel j'ai eu des discussions techniques toujours fructueuses.*
- *Monsieur M. JACOLIN qui, par son esprit critique, m'a permis de progresser et de réfléchir sur mon travail.*

Tous mes collègues de Laboratoire : Messieurs DEVILLERS, FINET DE LAMBERTERIE, REY, VIDART, ZHIRI, qui par leurs suggestions, leurs conseils ou leur contribution active m'ont permis de mener à bien ce travail.

Je tiens à remercier :

Les services de dactylographie et de tirage à qui je dois la réalisation matérielle de cet ouvrage.

La compagnie IBM pour son aide financière.

P L A N

Introduction 1

I - RAPPELS RELATIFS AUX OUTILS D'AIDE A LA CONSTRUCTION ET A LA MAINTENANCE DE SYSTEMES 7

I-1 Langage de haut niveau 7

I-2 Documentation 8

I-3 Détection des erreurs, mise au point 10

I-3-1 Dans le cadre des langages machine 10

I-3-2 Dans le cadre d'un langage de haut niveau 11

I-3-2-1 Génération de tests par le compilateur 11

I-3-2-2 Outils d'aide à la mise au point 12

I-4 Conclusion 16

II - OUTILS HOMOGENES D'AIDE A LA PROGRAMMATION 17

II-1 Le langage hôte 18

II-2 Intégration des modules 20

II-3 Mise au point assistée 26

II-3-1 Les fonctions de mise au point 26

II-3-1-1 Contrôle symbolique de l'exécution 26

II-3-1-2 Modification affichage calcul immédiat . 29

II-3-1-3 Localisation, extension, servitude, annulation 30

II-3-2 Caractéristiques particulières des fonctions de mise au point 32

III - PRESENTATION D'UNE IMPLEMENTATION 35

III-1 Rappel du cadre de fonctionnement 35

III-1-1 Système 'SPY' 35

III-1-2 Machine unique 41

III-2 Information fournie par le compilateur 43

III-3 Chargement assisté 51
III-3-1 Spécifications externes 51
III-3-2 Commandes, cartes de contrôle, erreurs 54
III-3-2-1 Commandes 54
III-3-2-2 Cartes de contrôle 55
III-3-2-3 Erreurs 56
III-3-3 Logique interne et résultats annexes 58
III-4 Mise au point assistée 63
III-4-1 Idées générales 63
III-4-2 Identification 66
III-4-2-1 Localisation du contexte 66
III-4-2-2 Calcul de l'adresse 67
III-4-2-3 Affichage et modification de variables 68
III-4-3 Machine de contrôle 73
III-4-3-1 Le contrôle de l'exécution 74
III-4-3-2 Table de contrôle et de fonctions attachées aux points de contrôle 74
III-4-3-3 Fonctions attachées aux points de contrôle 78
III-4-3-3-1 Fonctions du système d'aide à la mise au point 78
III-4-3-3-2 Fonctions accessibles à l'utilisateur 79
III-4-3-4 Forme externe des requêtes de contrôle accessibles à l'utilisateur ... 80
III-4-3-4-1 Les requêtes de contrôle ponctuel 80
III-4-3-4-2 La trace 81
III-4-4 Commandes diverses 83
III-4-4-1 Annulation et abandon 83
III-4-4-2 Enquête, Groupe de commandes, Reprise de l'exécution 85
III-4-5 Technique en vue d'une implémentation efficace 87

IV - CRITIQUE DU SYSTEME PROPOSE ET EVOLUTION FUTURE 97

IV-1 Comparaison avec des systèmes existants 97
IV-2 Environnement du Système de mise au point 102
IV-3 Quelques problèmes et les limites des outils 103
IV-3-1 Problèmes soulevés par la gestion des registres et l'optimisation 103
IV-3-2 Restrictions 114
IV-3-3 Extensions "hardware" 116
IV-4 Système d'aide à la programmation 118

Conclusion 121

Annexes 125

Annexe A - Rappels sur les grammaires de type 3 et schéma de l'analyse 125

Annexe B - Grammaire des commandes d'aide à la mise au point 133

Annexe C - Automates correspondant aux principales commandes 137

Bibliographie 143

- OUTILS DE MISE AU POINT POUR LANGAGES DE HAUT NIVEAU : ASSOCIATION DE MODULES ET CONTROLE DE L'EXECUTION -

INTRODUCTION

L'évolution de l'informatique et son utilisation de plus en plus intense amènent la création de systèmes complexes à la fois par leur taille et leurs fonctions.

La construction de tels systèmes soulève de nombreux problèmes; ceux-ci sont d'ordres très divers.

Si la taille du système est importante et nécessite un travail d'équipe, des problèmes relatifs à la gestion du projet apparaissent [T2]. Les difficultés sont engendrées par la coordination des différentes personnes ou équipes, par la définition et la distribution des tâches.

Le temps passé à la construction du système représente un facteur non négligeable. Son influence est souvent prédominante dans le calcul du coût, ainsi que dans l'originalité du système projeté, lors de sa mise en exploitation. Son estimation avant l'écriture du système n'est pas une tâche aisée.

Durant son exploitation normale, le système doit fonctionner sans erreur, quel que soit le contexte dans lequel il est placé. Cet aspect, extrêmement important, est généralement recouvert par le terme de "fiabilité". Les causes d'un manque de fiabilité sont diverses [T3], souvent liées à une insuffisance des tests ou à une mauvaise construction du système projeté.

Le manque de fiabilité est le plus grave reproche qui puisse être fait à un système; ses conséquences sont d'une part une utilisation pénible du système, d'autre part des erreurs souvent irréparables.

Indirectement liés à la construction on rencontre les problèmes relatifs à la documentation du système, aux modifications, à la portabilité, à l'efficacité et à la maintenance. (Nous désignons sous le mot maintenance l'ensemble des travaux qui sont effectués sur un système alors qu'il est dans sa phase d'exploitation. Il s'agit soit d'entretenir le système, de corriger les erreurs, soit de l'adapter en vue de nouvelles applications, ou de l'améliorer).

Parmi tous ces problèmes,ceux relatifs aux erreurs de fonctionnement sont perçus comme cruciaux même par des personnes peu expérimentées en informatique. Les erreurs de fonctionnement se manifestent par un comportement anormal du système, par rapport au comportement prévu par le programmeur. Parfois même, de telles erreurs peuvent apparaître au cours de la phase d'exploitation.

Si l'on tient compte de la remarque de DIJKSTRA [D5] "Program testing can be used to show the presence of bugs, but never to show their absence !", la solution à une fiabilité parfaite ne peut être obtenue que par la preuve formelle du fonctionnement correct des programmes [E2]. A la limite, la solution est dans l'utilisation de système effectuant des vérifications d'invariants [K1], ou assistant l'écriture des programmes [F2].

Toutefois, une assez bonne fiabilité peut être obtenue en utilisant des méthodes de programmation et d'aide à la construction de système.

Parmi ces méthodes,citons pour mémoire :

. La programmation modulaire [P1][P2]. Elle consiste à diviser le système en éléments (procédure, module, blocs, etc...),les éléments communiquant entre eux par des liens (variables globales, paramètres, etc...).

. La structuration du système en couches ou niveaux. L'exemple le plus connu est le système "THE" [D4].

. La programmation structurée [D1][D5][M2]. Elle fournit une méthode pour l'écriture et l'organisation des programmes.

. La programmation de haut en bas ("Top down programming") [M1]. Elle apparaît plutôt comme une méthodologie pour la structuration de système ; son utilisation est couplée avec celle de la méthode précédente.

L'utilisation de ces méthodes apporte plus de clarté dans l'écriture des programmes, facilite les modifications, permet une localisation plus aisée des erreurs et diminue les risques d'erreurs sans toutefois les annihiler.

On a alors recours :
- à des tests [E1], afin de constater un mauvais fonctionnement éventuel du système.
- à des outils, afin d'aider le programmeur dans la recherche des causes exactes des erreurs.

Les méthodes, les tests et les outils sont alors trois éléments complémentaires contribuant à la création de systèmes fiables.

Nous nous sommes plus particulièrement intéressés à la détection des erreurs de fonctionnement pour des systèmes écrits en langage de haut niveau.

Les erreurs de fonctionnement ont plusieurs origines :
- une mauvaise programmation. Ceci est le cas par exemple d'une absence de méthode lors de l'écriture des programmes.
- une méconnaissance du langage. Ce type d'erreurs est généralement détecté par le compilateur.
- une méconnaissance ou le non respect des liens unissant les modules (parfois appelé erreurs d'interface).
- une erreur dans la conception ou dans la programmation d'un algorithme.

Notre objectif a été l'étude et la réalisation d'outils aidant le programmeur dans la détection d'erreurs appartenant aux deux derniers types. Ces outils ont été réalisés pour des systèmes écrits en langage GSL [B5]. Ce travail fait suite aux travaux de MM. P. LEFEBVRE et B. PETEUL, relatifs aux outils permettant la mise au point de programmes écrits en langage machine [L3] [P3].

L'utilisation du cadre de travail fourni par un langage de haut niveau va apporter des solutions différentes à des problèmes qui se posent de manière analogue dans le cadre d'un langage machine. En effet, les renseignements que l'on peut obtenir à partir du code source d'un programme écrit en langage de haut niveau donnent des possibilités plus étendues que celles qu'il est permis d'espérer dans le cadre d'un programme écrit en langage machine.

Nous avons étudié et réalisé deux outils :

- une fonction d'intégration qui, lors du chargement des modules d'un système écrit en langage GSL, vérifie la cohérence des liens unissant les modules. (Nous qualifions aussi de manière abusive cette fonction de chargement assisté).
- un outil d'aide à la mise au point de programmes écrits en GSL.

Le fonctionnement de ces deux outils ne nécessite pas de modification des programmes objets engendrés par le compilateur GSL.

Ces outils sont interactifs; ils permettent ainsi un contrôle continu et une observation instantanée de la part du programmeur.

Le dialogue entre les outils et le programmeur se fait au niveau du langage source.

Le compilateur a été modifié; il fournit une description des objets manipulés et de la structure du système en construction. De plus, comme nous le verrons dans le chapitre 3, ces outils connaissent la configuration de la mémoire où réside le système en construction. Ainsi, ils sont capables de vérifier la cohérence des requêtes émises par l'utilisateur.

L'exploitation des renseignements nécessaires au fonctionnement de ces deux outils peut donner naissance à un ensemble d'outils d'aide à la programmation et à la maintenance de systèmes. On possède en effet une description de la structure du système en construction, des objets qui y sont manipulés etc...

Cet ensemble d'outils et ces renseignements complétés par d'autres informations (description fonctionnelle des modules, date d'écriture, etc...) peuvent donner naissance à un système d'aide à la gestion de projet.

Dans les chapitres suivants, nous donnerons un aperçu des outils d'aide à la programmation existants. Nous montrerons ensuite le contexte dans lequel nous travaillons, les aspects fondamentaux des outils proposés et la justification de certaines orientations. Nous décrirons les traits principaux d'une implémentation et les difficultés soulevées par la résolution de certains problèmes.

On pourra en effet remarquer que les problèmes ne sont pas spécifiques au cadre du langage GSL, mais sont relatifs à la construction d'outils similaires destinés à d'autres langages de haut niveau.

En dernier lieu, nous verrons les limites auxquelles nous nous sommes heurtés et les extensions qu'il est possible de donner à ce projet.

CHAPITRE I

RAPPELS RELATIFS AUX OUTILS D'AIDE A LA CONSTRUCTION ET A LA MAINTENANCE DE SYSTEMES

Au cours de ce chapitre, nous verrons des outils fournissant une aide à l'utilisateur pendant les phases du développement d'un projet "software".

D'une manière très générale, les buts que ces outils se proposent d'atteindre sont :

- Minimiser le coût de la production de "Software" en diminuant le temps nécessaire à l'exécution des différentes phases de la construction d'un système (phase de conception, d'écriture, de mise au point, etc...)

- Apporter une aide lors de la maintenance d'un système et lors de son évaluation [P4].

Nous observerons ces outils en estimant leur souplesse d'emploi, ainsi que leur sûreté, c'est-à-dire leur aptitude à détecter ou à éviter de créer de nouvelles erreurs.

I-1 - LANGAGE DE HAUT NIVEAU

L'utilisation d'un langage de haut niveau apporte une aide efficace à la construction de système. En effet, l'utilisation d'un langage de haut niveau :

- permet à l'utilisateur de se dégager des contingences des langages machine pour ne s'attacher qu'à la description des algorithmes.

- fournit un moyen de structurer et d'établir des relations entre les données d'un programme. (ex : utilisation de variables dimensionnées, ou référencées par l'intermédiaire d'une variable de type référence à un mode).

- fournit des fonctions pour manipuler ces données.
(ex : fonctions permettant les calculs arithmétiques).

- diminue le temps passé à l'écriture.

- facilite la compréhension des programmes, ainsi que la mise en oeuvre de méthodologies telles que la programmation structurée ou modulaire.

Ces points concernentdes facilités d'emploi ; certains langages, tout en conservant les avantages précédemment énoncés, présentent des aspects permettant de limiter les risques d'erreur.

Cette faculté est due à :

- Une syntaxe et une sémantique bien conçues , c'est-à-dire qui permettent de définir de manière précise les objets et les manipulations qui portent sur ces objets. Une illustration de ce point est la définition du langage ALGOL 68 [W1].

- Des vérifications lors de la phase de compilation. Le compilateur est capable de détecter une classe d'erreurs. En particulier, il vérifie l'utilisation correcte des objets manipulés en fonction de leur déclaration et de l'environnement au point d'utilisation (niveau de bloc, position de "coercion", limites de variation). Cependant, certaines de ces vérifications ne peuvent être faites que lors de l'exécution ; de plus, elles dépendent du langage considéré. Les compilateurs des langages PASCAL [W2] ou GSL [B5] présentent de tels avantages.

I-2 - DOCUMENTATION

La conduite d'un projet software nécessite une organisation efficace du point de vue humain [T2]; l'implémentation, la mise au point ainsi que l'entretien de systèmes importants nécessitent de connaître les imbrications ou les dépendances existant entre les composants ou les objets manipulés dans le système. Fréquemment, cette tâche est accomplie par les membres de l'équipe ; il y a alors création,

exploitation et mise à jour manuelle des documents décrivant la structuration du projet. Cette méthode s'avère lourde, fastidieuse ; de plus elle est sujette à l'humeur des personnes travaillant à la réalisation du projet.
Il est intéressant de posséder des outils capables de collecter, puis d'exploiter les informations utiles à la compréhension de la logique du système globalement ou localement au niveau du module. A la limite, ces outils sont capables de gérer les différents stades de l'avancement du projet ainsi que de coordonner les travaux des diverses équipes.
Ces outils peuvent être divisés en deux classes, suivant les possibilités qu'ils offrent.

- La première classe inclut les outils permettant une meilleure compréhension de la logique du système en construction ou en cours de maintenance. Les aspects de l'aide apportée par ces outils sont :

  - Une standardisation des présentations des programmes. [C3](restructuration de la forme des listes de sortie, mise en évidence de certains types d'instructions).

  - Une amélioration des recherches auxquelles le programmeur peut être amené, par l'élaboration de tables des références croisées au niveau des modules (cette fonction est généralement incorporée au compilateur) ou au niveau global de l'ensemble des modules [T1].

  - Une compréhension meilleure et plus rapide de la logique du système par la génération automatique des organigrammes de chacun des modules [L4], ou par la génération du graphe des appels des composants du système.

  - Une estimation statique du coût de l'exécution par l'intermédiaire de l'évaluation probabilistique du graphe du programme.

- La deuxième classe inclut les outils permettant une gestion automatique du projet [C2] [G1].
Ce type d'outils fournit l'espace nécessaire pour stocker les composants du système avec leurs descriptions et donne un moyen aux différentes équipes du projet de construire, d'intégrer, de tester la partie qui leur incombe sans interférer avec les autres équipes et de connaître l'état de l'avancement du projet.

L'aide apportée par ces outils est indéniable ; cependant leur fiabilité diminue lorsque les informations nécessaires à leur fonctionnement sont collectées par l'intermédiaire d'une enquête. En effet, dans ce cas, l'outil devient tributaire des réponses erronées que peut fournir l'utilisateur.

I-3 - DETECTION DES ERREURS, MISE AU POINT

Durant cette phase du développement d'un projet, le programmeur s'attache à vérifier que le système écrit fonctionne correctement et fournit les résultats escomptés. Ce paragraphe s'intéresse à la description d'outils permettant à l'utilisateur, son programme s'exécutant dans un environnement de tests, d'en observer et d'en modifier le comportement, d'être renseigné lorsque des conditions anormales surviennent. Le choix judicieux des tests reste à la charge du programmeur [P5].

Une anthologie de la mise au point est fournie par STOCKTON GAINES [S5]; nous nous bornerons pour notre part à classer les outils d'aide à la mise au point existants,à mettre en relief les traits essentiels qui les caractérisent.

1-3-1 - Dans le cadre des langages machine

Au niveau des langages machine, le nombre d'outils d'aide à la mise au point est important ; en effet, dans bien des cas, leur but initial fut la mise au point du système d'exploitation fourni à l'utilisateur [B9] [I1]. Une fois le système opérationnel, l'outil de mise au point y a été maintenu en tant que composant utilitaire et parfois aussi en tant que routine de sortie sur fin anormale de programme (utilisateur ou système).

Leurs possibilités sont restreintes : manipulation d'adresses machine ou d'équivalents symboliques. Ce manque de puissance a amené le développement d'outils plus élaborés [I2] [I3] [I7] [L3] [M4] [P3] offrant des possibilités

de travail à un niveau symbolique, d'arrêts sur condition, de trace, de simulation [K2], de "patch" [E3]. Cependant, la mise au point d'un programme écrit en langage évolué avec ces outils reste malaisée, manque de fiabilité et oblige le programmeur à connaître les caractéristiques de génération du compilateur ; le code machine issu d'une compilation optimisée peut ne plus correspondre instruction par instruction au programme source.

1-3-2 - Dans le cadre d'un langage de haut niveau

Nous considérons les aspects que peuvent revêtir les outils d'aide à la mise au point, dans le cadre d'un langage de haut niveau. L'intérêt de base présenté par de tels outils est de permettre à l'utilisateur d'ignorer totalement les mécanismes sousjacents à l'exécution du code source (forme de la génération, interface avec le système d'exploitation ou avec le superviseur d'exécution nécessaire à l'exécution du programme).

1-3-2-1 - Génération de tests par compilateur

Un programme comporte des points critiques ; ce sont par exemple les utilisations d'une tranche de tableau, ou d'une sous-chaîne, ou encore les transferts de contrôle [A1].

La génération de tests par le compilateur a pour but de vérifier, lors de l'exécution, le passage correct de ces points critiques.

- La syntaxe et la sémantique du langage donnent au programmeur la possibilité de spécifier les conditions critiques qu'il désire détecter et traiter. C'est le cas du langage PL/1 [I4] ou de ses dérivés [C4] (ex : ON OVERFLOW, ou ON SUBSCRIPTRANGE).

- Sur option, le compilateur produit les séquences (code exécutable ou interprété) nécessaires à la détection et au traitement des conditions anormales.

Cette deuxième méthode s'avère moins souple que la précédente ; elle ne permet pas à l'utilisateur de contrôler les erreurs mais seulement d'en avoir connaissance.

Souvent les options prises par défaut pour la poursuite de l'exécution ne correspondent pas au désir du programmeur.

De plus, la taille du code généré est accrue, et en cas d'erreurs survenant dans un programme en exploitation normale, cette méthode implique la recompilation du système avec l'option de test.

Dans la détection des erreurs et dans la clarté des messages, citons comme cas extrême le compilateur DITRAN [M3], mais dont l'emploi a surtout un but pédagogique.

La génération des tests est réalisée dans les deux cas durant la compilation ; ainsi on dispose de toute l'information décrivant les éléments incriminés.

1-3-2-2 - Outils d'aide à la mise au point

a) Schéma général

Le but de cette catégorie d'outils est de donner au programmeur la possibilité de détecter des conditions anormales de fonctionnement non seulement en des points critiques connus, mais aussi en tous points du système.
Ces outils présentent entre eux des similitudes au niveau de leurs fonctions [J1] et de leur structuration externe ; le schéma suivant se retrouve dans chacun d'eux sous une forme plus ou moins simplifiée.

- La communication avec l'utilisateur se fait par l'intermédiaire d'un langage subdivisé en deux parties :

  - La première sert à exprimer le contrôle de l'exécution (ex: trace, localisation) et les points critiques où il a l'intention de faire des tests. Ces arrêts peuvent être inconditionnels (le système à mettre au point perd le contrôle de l'exécution au profit du système servant à faire les tests. C'est le cas de la commande "AT" de FORTBUG [J2] par exemple) ou conditionnels (la reprise du contrôle n'a lieu que lorsque des événements particuliers surviennent. C'est le cas de la commande "WHEN" dans le système SPY [L3]).

- La seconde partie offre la possibilité de localiser et de manipuler les éléments du système à tester (Affichage, modifications, etc...).
  Cette partie possède une intersection commune avec le langage d'écriture des programmes du système. Ceci permet de travailler avec les mêmes structures de données que celles autorisées dans le langage [G4].

  Les outils dits de "dump post mortem" [B3] [N2] sont un sous-ensemble des outils d'aide à la mise au point, la partie contrôle de l'exécution ayant été supprimée. La seconde partie (manipulation d'objets) se trouve parfois spécialisée au profit de la mise au point d'un système particulier.

- Des fonctions de contrôle transparentes à l'utilisateur permettent à l'outil de test de suivre le déroulement du programme (ex: établissement de l'adressabilité dans la pile pour un langage type ALGOL) et d'établir les liens avec le système d'exploitation gérant le calculateur.

b) Types d'outils

Il apparaît deux classes d'outils d'aide à la mise au point suivant le type du système d'exploitation sur lequel on travaille.

- Les outils "batch" : le programmeur doit spécifier tous les tests à accomplir avant l'exécution des programmes [I6] et parfois avant leur compilation [I5].
- Les outils interactifs : le programmeur a la possibilité, par l'intermédiaire d'un terminal, d'observer et de contrôler l'exécution du système [G3] [J1].

Une polémique oppose les partisans des deux classes d'outils ; un outil interactif s'avère supérieur car il permet au programmeur d'intervenir en fonction de l'état immédiat de son programme ; cependant, il favorise un manque de méthodes dans l'organisation des tests.
Une comparaison quantitative des deux classes a été faite par SACKMANN et ERICKSON au niveau des coûts [S1] ; il en ressort qu'un outil interactif

nécessite moins de"temps homme"pour la mise au point d'un même système qu'un outil batch, mais accroit le temps machine nécessaire.

c) Techniques employées

Donnons un rapide aperçu des techniques employées afin de résoudre les deux difficultés principales que présente la construction d'un outil d'aide à la mise au point, à savoir :

- Le contrôle de l'exécution.
- L'établissement des liens entre la forme externe des objets manipulés et leur forme interne.

Ceci permet de mieux saisir les limites des outils d'aide à la mise au point.

- L'interprétation employée en particulier dans les compilateurs incrémentaux [B4] [G3] [U1] présente une grande souplesse (modification des lignes source, contrôle de l'exécution, recherche des éléments manipulés). Cependant l'exécution est ralentie dans des proportions non négligeables. La compatibilité entre un programme compilé en code machine exécutable et un programme interprété existe très rarement ; ceci implique que lors d'une mise au point, tout le système doit être interprété. La compatibilité pose quelques sérieux problèmes, au niveau de l'accès à la pile d'exécution dans le cas d'un langage du type ALGOL 60, au niveau des variables communes pour des langages du type FORTRAN ; cela nécessite la conception du compilateur, de l'interpréteur et du superviseur d'exécution de manière synchrone.

- L'insertion de code consiste à générer lors de la compilation pour chaque instruction du texte source, une séquence de transfert vers un superviseur d'exécution [I7] [J1]. Cette méthode ne résoud pas les problèmes de la manipulation des objets ; elle accroît la taille du code exécutable provenant du texte source ainsi que le temps d' d'exécution ; elle présente l'avantage de conserver une compatibilité totale entre les modules ainsi modifiés et ceux compilés normalement.

- Modification du texte source.

  Un préprocesseur traite les fonctions de tests définies par l'utilisateur au moyen du langage de communication et les inclut dans le texte source. Le programme source ainsi modifié est ensuite compilé [D1] [F1]. Les inconvénients de cette méthode sont un accroissement du code généré, l'obligation de recompiler un module pour toute modification apportée aux ordres de test l'intéressant. Les problèmes de compatibilité et de manipulation d'objets du système à tester sont résolus avec un coût minimum.

- Substitution dans le code exécutable.

  La reprise du contrôle s'effectue en substituant (dans le code compilé du programme à tester) une séquence de transfert vers l'outil d'aide à la mise au point, la ou les instructions substituées étant ensuite simulées ou exécutées hors de leur contexte initial [S4]. Cette méthode ne modifie pas la taille du système contrôlé ; elle dépend totalement du type de calculateur sur lequel les programmes s'exécutent.

Après ce bref survol, on voit apparaître les difficultés engendrées par l'utilisation d'outils d'aide à la mise au point :

- Augmentation du temps d'exécution.
- Augmentation du code généré (critique dans le cas des langages destinés à l'écriture de systèmes d'exploitation).
- Souplesse d'utilisation (coûts des modifications des tests prohibitifs ou variables définissant l'état du programme inaccessibles).
- Liaisons entre identificateurs et variables (nécessitent des vérifications d'existence pour certains langages).

Trois nouveaux problèmes viennent se greffer :

- Fiabilité : l'outil doit parer, lorsqu'il le peut, à toute erreur de l'utilisateur.
- L'outil doit être mis hors d'atteinte des programmes testés.

- Dans le cas d'un compilateur optimiseur, le programme généré n'est plus la traduction exacte instruction par instruction du programme source ; de plus, les instructions ne sont plus indépendantes entre elles.

1-4 - CONCLUSION

Les gains apportés de façon globale en programmation par les outils décrits sont indéniables; cependant, dans la majorité des cas, les efforts nécessaires à leur mise en oeuvre rebutent l'utilisateur, leur fiabilité est directement liée à la "bonne conscience" du programmeur.

Pour créer les informations utiles au fonctionnement de ces outils, fréquemment le travail est fait de manière redondante. Ces informations sont en effet souvent disponibles puis perdues au cours d'une phase précédente.

CHAPITRE II

OUTILS HOMOGENES D'AIDE A LA PROGRAMMATION

Ce chapitre a pour objet la présentation des traits généraux de deux outils d'aide à la programmation :

Une fonction d'intégration (ou chargement assisté) et un outil d'aide à la détection des erreurs.

Bien que l'implémentation ait été faite pour le compte d'un langage particulier (GSL [B5]), les aspects et les problèmes se retrouvent lors de la construction de tous outils similaires quel que soit le langage.

Le but de ces deux outils est d'aider à la détection d'erreurs contenues dans un système en construction écrit en GSL,

L'information nécessaire à leur fonctionnement est commune.

Nous justifierons aussi au cours de ce chapitre les orientations que nous avons prises.

Avant de passer à une description des outils proposés, justifions l'emploi du terme "homogène" qui les qualifie.

Nous avons voulu exprimer que :

- Quel que soit l'outil considéré, la communication avec l'utilisateur s'effectue de manière symbolique. Les objets manipulés sont ceux qui composent ou appartiennent au système en construction.

- Quelle que soit la phase considérée (phase d'écriture, intégration, mise au point), on essaie de fournir un outil adéquat.

II-1 - LE LANGAGE HOTE

Nous appelons "langage hôte" le langage dans lequel sont écrits les programmes du système en construction.

Pour notre cas, il s'agit du langage GSL [B5]. Ce dernier est destiné à l'écriture de systèmes (système d'exploitation, compilateur, etc...) et permet de libérer le programmeur des contingences du langage machine, tout en conservant un contrôle étroit du calculateur [L2].

L'augmentation de la fiabilité (détection d'erreurs à la compilation) et de la clarté des programmes, procurée par son utilisation, permet de considérer le langage GSL comme un outil d'aide à la construction de système.

Les caractéristiques du langage ont aiguillé certains choix d'implémentation, en particulier celui relatif à la non modification du code objet engendré par le compilateur du langage GSL.

a) Caractéristiques du code objet

La compilation d'un texte source GSL donne naissance à un texte objet possédant les caractéristiques suivantes :

- Le code objet est du code machine exécutable et translatable.
- L'exécution du code objet s'effectue sans le secours d'un superviseur d'exécution (Superviseur d'entrée sortie, d'allocation mémoire, etc...).
- Le code objet ne contient que les variables apparues dans le texte source. Le compilateur ne crée pas de variables de travail (de manière classique, les compilateurs génèrent des variables de travail pour évaluer des expressions, ou pour les appels de procédure par exemple).

Ces deux derniers points permettent de contrôler très exactement la place qu'occuperont des programmes en mémoire. Dans des langages de haut niveau tels que PL/1 ou ALGOL 60, ces mécanismes sont partiellement cachés. Ceci vient des buts assignés aux langages ; GSL étant orienté vers la programmation système, les motivations diffèrent.

b) Modularité et mécanismes de liaison

Comme nous l'avons vu sommairement en introduction, il existe des méthodes de construction de système. Elles préconisent toutes l'utilisation de modules. (Les avantages principaux de la programmation modulaire sont une plus grande facilité et rapidité de mise au point, de modification, de compréhension, etc...)

Le langage GSL offre la possibilité de programmer de manière modulaire ; le module étant actuellement à la fois l'unité de compilation, d'édition et la procédure la plus externe.

Contrairement aux langages PL/1, ALGOL ou FORTRAN où les mécanismes de liaison entre procédures sont figés et cachés au programmeur, en GSL ils peuvent être totalement spécifiés lors de la déclaration de la procédure.

Les échanges d'informations entre modules s'effectuent par l'intermédiaire de paramètres ou de zones communes.

c) Vérification

En vue de ne pas accroître le temps d'exécution et la taille du code objet, les vérifications dynamiques ne sont pas incluses dans le code objet. Par ailleurs, un maximum de vérifications (statiques) sont réalisées durant la phase de compilation. Ainsi, par exemple, le mode d'une procédure est défini par le mode de ses paramètres, le mode de ses résultats et celui des éléments appartenant à l'environnement.

d) Contrôle de la machine

Le programmeur peut, s'il le désire, attacher un nom et un mode aux registres machine, spécifier qu'il gère ces registres sur certaines portion de programme ou laisser ce soin partiellement ou totalement au compilateur.

D'autre part, le langage permet l'insertion de séquences de code machine dans le programme source.

e) Efficacité du code objet

Bien que fonctionnant en un seul passage, le code objet est optimisé. L'optimisation a lieu au niveau de l'utilisation des registres ; ces derniers peuvent être réutilisés directement s'ils contiennent une valeur de variable par exemple [C1]. Cet aspect est important : il détruit l'indépendance des instructions source entre elles ou la correspondance biunivoque entre une instruction source et le groupe d'instructions du code objet.

Cette brève description fait ressortir les points suivants :

- Contrôle étroit du calculateur.
- Maximum de vérifications avant l'exécution.
- Optimisation du code objet engendré.

Les caractéristiques et l'implémentation des outils d'aide à la programmation en GSL ne doivent pas rendre caduc l'un de ces points.

II-2 - INTEGRATION DES MODULES

Nous nommons ainsi les fonctions permettant, à partir des modules jusqu'ici traités séparément, d'intégrer ces derniers pour en faire un seul objet.

Cette phase d'intégration ayant lieu au cours de la phase de chargement, nous avons improprement appelé le tout "chargement assisté".

Nous retrouvons donc deux fonctions principales et disjointes du point de vue logique de fonctionnement :

- Une fonction de chargement et "d'édition de liens".
- Une fonction de vérification ou d'intégration qui vérifie la cohérence des liens au niveau du code source des modules.

a) Fonction de chargement et d'édition de liens

Cette fonction est classique [P6]; elle amène en mémoire les modules objets engendrés lors de la compilation et résoud les liens entre eux au niveau des adresses. Elle constitue ainsi un système exécutable en mémoire.

Les possibilités de cette fonction seront vues en détail au cours du chapitre suivant. Signalons toutefois :

- qu'elle tient compte du mécanisme de segmentation de CP [I8] et des mémoires dites à "trous" [L1].
- qu'elle permet de séparer le code des données, lors du chargement des modules objets issus de la compilation de programmes GSL.
- qu'elle permet de simuler un chargeur d'IPL ("Initial Program Loading") en utilisant le support SPY [H1] [L3].

b) Fonction de vérification ou d'intégration

Cette deuxième fonction permet de qualifier le chargement d'"assisté" elle rend l'intégration des modules écrits en GSL plus fiable.

En effet, chaque module constituant une unité pour la compilation est traité séparément des autres modules. Cette indépendance disparaît lors de l'exécution, par suite d'échanges d'informations s'effectuant par l'intermédiaire d'éléments de liaison.

Ces éléments de liaison sont :

- Soit des zones communes appartenant à un module (déclarées avec l'attribut "ENTRY" dans le module) et pouvant être référencées à partir d'autres modules (elles apparaissent alors déclarées avec l'attribut "EXTERNAL").

- Soit des paramètres passés lors de l'appel d'une procédure connue dans un module. L'en-tête de la procédure appelée (paramètres, résultats, type de l'appel, etc...) est alors répétée dans la procédure appelante.

Pour engendrer le code objet correspondant à la manipulation d'un tel élément, le compilateur se sert de la déclaration locale apparue dans le module. Si cette déclaration est entachée d'erreurs (elle n'est pas la description exacte de l'élément), l'exécution du code objet est alors susceptible de ne pas accomplir ce que le programmeur désirait effectuer.

La fonction de vérification a pour but de contrôler l'identicité ou la compatibilité des modes attachés par différentes déclarations d'un même élément de liaison, apparues dans divers modules du système en cours de construction.

Un problème très proche de celui-ci s'est posé lors de la réalisation du compilateur ALGOL 68 R, où un découpage des programmes en segments, compilés indépendamment, introduit la notion d'éléments globaux à plusieurs segments [C5] .

D'une manière générale, ce problème se pose dans tout langage où des variables sont communes à plusieurs modules compilés séparément : c'est le cas, par exemple, de FORTRAN et de PL/1.

Donnons un exemple comportant des déclarations erronées :

```
A :  PROC (R1  REG (1)  FIXED) ; déclaration de la procédure A;
     |                           implicitement elle porte l'attribut "entry"
     |
DCL  CH  CHARACTER (10) EXTERNAL ;  }
                                    }  déclarations locales.
DCL  B  PROC  EXTERNAL ;            }
     |
     |
 CH = 'OA'X ;  affectation de OA16 dans le premier caractère de la chaîne
     |         CH; dans les 9 autres caractères suivants, on place le code
     |         du caractère blanc.
     |
CALL  B ;
     |
     |
END ;
```

```
B :  PROC  ENV (R3  REG (3) PTR) ; déclaration de la prócédure B; R3
                                   appartenant à l'environnement de la
                                   procédure ne doit pas être détruit.

DCL  A  PROC (REG (2)  FIXED) EXT ; déclaration locale de la procédure A,

DCL  CH  BYTE (10)  ENTRY ; CH appartient à la zone des données de la
                                   procédure B.

DCL  BZ (10)  BASED (R3) ;

CALL  A (256) ;    appel de la procédure A; la valeur 256 est placée
                   dans le registre 2 paramètre de A comme l'indique la
                   déclaration locale.

END ;
```

Dans cet exemple, on peut relever les incohérences suivantes :

- Pour la même procédure A, les deux déclarations donnent des modes différents (le mode d'une procédure est défini par le mode de ses paramètres, celui de la liste de retour et de la liste d'environnement). Lors de l'appel de A dans la procédure B, la valeur 256 est placée dans le registre deux, le registre un contiendra lors de l'exécution de A une valeur quelconque.

- La variable CH sera manipulée dans A comme une chaîne de caractères, dans B comme une chaîne d'octêts. Ceci peut entraîner des erreurs, en particulier lorsqu'une sous-chaîne de CH est complétée au cours d'une affectation.

- Lors de l'appel de B, le registre R3 est sensé contenir une adresse. Il servira à adresser le tableau BZ ; sa valeur dans A étant quelconque, il peut s'en suivre des erreurs à l'exécution.

La fonction de vérification détectera ces incohérences, lorsque ces deux modules seront chargés ensemble.

On voit qu'elle permet ainsi de compléter les vérifications accomplies par le compilateur, à l'intérieur de chacun des modules, au niveau de la cohérence des utilisations d'un objet et de sa déclaration.

c) Quand réaliser ces vérifications ?

L'intérêt de ces vérifications est peu discutable, sauf pour ceux dont le seul but en programmation est d'engendrer des effets de bord.

Le moment choisi pour accomplir ces vérifications peut être placé lors de la phase de compilation, de chargement ou d'exécution.

- On écarte le choix de la phase d'exécution. En effet, les vérifications augmenteraient le temps d'exécution du système en constructio De plus, la syntaxe du langage GSL permet d'éviter les vérifications de mode à l'exécution.

- Durant la phase de compilation, il est possible d'effectuer ces vérifications ; elles nécessitent un système de gestion de fichiers. Ce dernier a deux rôles : gérer différents niveaux d'un même module et vérifier la compatibilité des niveaux lors de l'intégration des programmes du système en construction (on suppose qu'il n'y a pas d'utilisations du type A utilise B et B utilise A). Supposons que A fait référence à des éléments contenus dans B et dans C ; lorsque l'on compile A, on spécifie au compilateur d'aller chercher ces éléments dans B et C. Si par la suite B ou C subissent des modifications, leur niveau change ; lors de l'intégration, des incompatibilités de niveau seront détectées.

Une telle méthode a été utilisée pour implémenter le compilateur "ALGOL 68 R" [C5]. Un programme est découpé en segments conservés dans un "Album"; la compilation d'un segment peut donner naissance, en plus du code objet, à une "Keep list", ensemble d'éléments pouvant être utilisés lors de la compilation d'un autre segment. A chaque segment est attaché un "magic number" mis à jour si une modification est intervenue dans le segment, et servant à vérifier la compatibilité lors de l'intégration.

Ne possédant pas de système de gestion de fichiers adéquat, sa constitution étant dans un premier temps trop coûteuse, il est apparu que l'exécution des vérifications lors du chargement était tout aussi efficace. On dispose en effet durant cette phase des noms de composants du système en construction et de toutes les informations nécessaires aux vérifications.

d) Caractéristiques propres au chargeur assisté

Chacune de ces fonctions présente un certain nombre de caractéristiques et fournit des résultats annexes au chargement.

- Fiabilité. L'utilisateur dirige le chargement à l'aide d'ordres spéciaux. La fonction de chargement vérifie la cohérence des ordres de chargement entre eux, la cohérence des ordres de chargement vis-à-vis de la configuration de la mémoire de la machine où s'effectue le chargement. Ce deuxième point est utile dans le cas d'une mémoire discontinue (dite à "trous") [L1]. Une erreur trop importante annule l'exécution du chargement ou interdit l'exécution du système chargé.

- Résultats. En plus d'un système exécutable en mémoire centrale, le chargeur assisté fournit :

  . Une description de l'implémentation en mémoire du système en construction (localisation du code objet translaté et des données).

  . Un graphe statique décrivant les appels des modules entre eux et l'utilisation des variables globales.

  . Une description des éléments translatables apparus dans le code objet des modules (l'adresse où ils sont référencés et leurs propriétés attachées par leur déclaration dans les textes sources des modules).

II-3 - MISE AU POINT ASSISTEE

Après la phase de chargement assisté, une nouvelle phase débute, celle dite de mise au point.

Le programmeur désire connaître le comportement des modules qui viennent d'être amenés en mémoire, savoir si les résultats correspondent à ceux escomptés, suivre le déroulement de l'exécution, etc...

Des fonctions spéciales assistent le programmeur pour réaliser ces tâches de mise au point.

Nous donnons dans ce paragraphe une description sommaire des différentes fonctions qu'il est possible de fournir (une description plus détaillée et une implémentation apparaitront au cours du chapitre suivant) ; on s'attache essentiellement à exposer les caractéristiques qu'elles possèdent et leurs justifications.

II-3-1 - Les fonctions de mise au point

II-3-1-1 - Contrôle symbolique de l'exécution

Le contrôle symbolique de l'exécution permet à l'utilisateur d'arrêter le déroulement de son programme ou d'en observer le passage au travers des modules du système en construction. Dans notre cas, à l'intérieur d'un module, tout objet manipulé est identifié par son nom symbolique ou par son nom symbolique et son numéro lorsqu'il s'agit d'une instruction.

a) Mécanisme d'arrêt

- Il permet d'arrêter l'exécution du système en construction.

Exemple : *Supposons que l'on veuille arrêter l'exécution du programme sur l'instruction 25 qui porte aussi l'étiquette ET1 ;*

On émettra la requête :

AT 25 STOP

ou la requête possédant le même effet :

AT TOTO STOP

L'arrêt peut être pondéré en demandant qu'il n'ait lieu que lorsque l'instruction aura été exécutée un certain nombre de fois.

- Le paramètre de la requête peut être un identificateur d'un type autre qu'étiquette ; l'arrêt aura lieu à chaque exécution des instructions où l'on fait référence à cet identificateur.

- On a essayé de reporter à l'exécution la mise en place des tests qui généralement sont incorporés dans le code objet par le compilateur.

Les tests sont faits lors de l'utilisation d'une tranche de tableau ou d'une sous-chaîne. L'arrêt a lieu lorsqu'une valeur, incorrecte de l'indice par rapport à la déclaration de l'élément, est détectée.

Dans tous les cas, une fois le contrôle rendu à l'utilisateur, l'émission de toutes les requêtes est possible.

b) Reprise de l'exécution

La reprise de l'exécution se fait séquentiellement. Cette option est justifiée par les problèmes que posent les vérifications pour traiter une requête "GOTO".

c) Mécanisme de trace

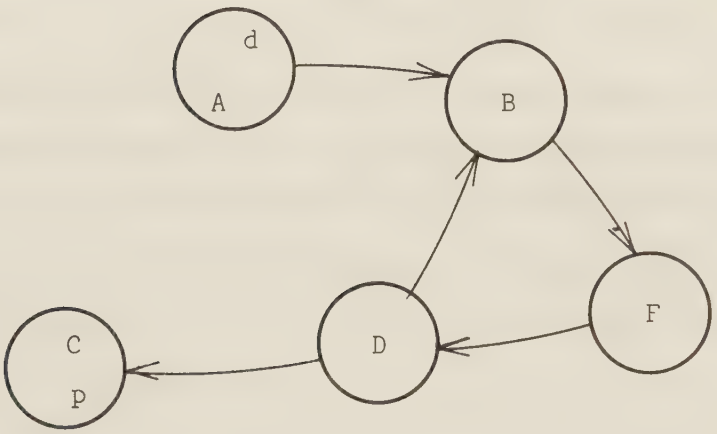
On possède deux mécanismes de trace :

. La trace des modules activés puis quittés au cours de l'exécution du système à mettre au point.

. L'ensemble des modules actifs à un moment donné.

Exemple : *Supposons que le système soit composé de cinq modules A, B, C, D et E, les flèches indiquant les appels (ce sont des appels fermés).*

Supposons que l'exécution aille du module A au module C en passant par les modules A, B, E, D, B, D, C.



On met la trace en fonction en d dans le module A (premier type de trace) ; on a alors les messages :

```
ENTER     B
ENTER     E
ENTER     D
ENTER     B
RELEASE   B
ENTER     C
```

L'utilisateur reprend le contrôle en p du module C, il demande la trace des modules actifs, il a alors les messages suivants :

C CALL By D
D CALL By E
E CALL By B
B CALL By A

Ce mécanisme de trace des modules actifs est réalisé par intermédiaire d'une pile, dont la mise à jour est faite à l'activation et à la désactivation de chacun des modules. Cette même pile sert à effectuer des vérifications, lorsqu'un utilisateur désire atteindre un module.

II-3-1-2 - Modification, affichage, calcul immédiat

Ces fonctions ont pour rôle de permettre l'affichage et la modification des variables apparues dans le système en construction.

Ces requêtes sont exprimées dans un langage, sous-ensemble du langage GSL (elles respectent les conversions, manipulent les mêmes structures de données, etc...).

Ces fonctions présentent un aspect fondamental : elles vérifient la cohérence de l'utilisation d'une variable par rapport à :

- Sa déclaration faite dans le texte source du module.

- La configuration de la mémoire de la machine utilisée ; de plus, elles vérifient la validité de l'accés relativement au point où le contrôle a été rendu.

Donnons des exemples d'utilisation de ces fonctions. Supposons qu'un programme GSL contienne les déclarations suivantes :

DCL P1 POINTER ;	*déclaration d'un pointeur local.*
DCP PARRAY (10) POINTER BASED ;	*déclaration d'un tableau de pointeurs dont l'adresse doit être fournie explicitement.*

DCL 1 STRUT (20) BASED, *déclaration d'un tableau dont chaque élément est une structure de la forme :*

2 ELM1 CHAR (20), - *Le 1er élément chaîne de 20 caractères*
2 ELM2 FIXED, - *Le deuxième nombre sur 4 octects*
2 ELM3 POINTER ; - *Le troisième un pointeur.*

- *affectation de la valeur 12000 en hexadécimal dans le 4ème élément du tableau PARRAY dont l'adresse est contenue dans P1 :*

P1 → PARRAY (4) = '12000'X

- *on émet la requête : P1 → PARRAY (12) = 256*
elle est refusée ; le tableau n'a que 10 éléments.

- *on émet la requête : P1 = - 256*
l'utilisateur est averti qu'il affecte une valeur incorrecte à un pointeur.

- *la requête :*

P1 → PARRAY (4) → ELM1 (8, 1 : 10) = 'TOTO'

modifie les caractères 1 à 10 du 8ème élément du tableau de structure dont l'adresse est contenue dans le 4ème élément de PARRAY.

- *la requête : PARRAY (2) = '12000'X est refusée, PARRAY étant une variable basée.*

Une demande d'affichage a la même forme. On veut connaître la valeur du 4ème élément de PARRAY dont l'adresse est dans P1 ; on émet alors la requête

P1 → PARRAY (4) ?

'12000' X réponse de la fonction d'affichage.

II-3-1-3 - Localisation, extension, servitude et annulation.

a) Qualification

Nous avons vu que les objets manipulés étaient repérés par un nom.

Or, des objets différents appartenant à des modules disjoints peuvent porter le même nom. Il faut donc un moyen permettant d'associer de manière univoque à un identificateur une des variables lui correspondant.

Ceci est réalisé en spécifiant un contexte [J1] c'est-à-dire une portion du système en construction où, à un identificateur, correspond une seule variable. Dans notre cas, la plus grande portion du système en construction où cette relation est biunivoque quel que soit l'identificateur choisi est le module.

Lorsque l'utilisateur spécifie qu'il désire atteindre tel module, on vérifie que le module qu'il désire atteindre est actif (mécanisme de trace) ; en effet, certaines variables peuvent ne pas être en vie.

b) Mécanismes d'extension

Deux mécanismes d'extension sont prévus :

- Le premier mécanisme donne la possibilité à l'utilisateur, lorsqu'il a le contrôle, d'activer des procédures préalablement chargées avec le système en construction.
  Cet aspect peut être utile pour observer l'état d'un fichier par exemple. L'utilisateur constitue ainsi ses propres fonctions de mise au point. Cependant, seule la cohérence de l'appel peut être vérifiée ; la bonne exécution du programme activé reste à la charge de l'utilisateur (le système de mise au point n'a aucune possibilité de s'apercevoir des effets de bord ou des modifications involontaires du système objet qui ont lieu).

- Le deuxième mécanisme permet de regrouper des requêtes de mise au point en des macro-requêtes. Leur activation peut être faite directement par l'utilisateur, ou automatiquement si la requête est attachée à un arrêt.

c) Fonctions d'annulation et de servitude

Le but de ces fonctions est d'annuler l'effet d'une requête précédemment émise ou d'aider l'utilisateur à se remémorer les fonctions actives. Ces requêtes ne présentent pas d'aspects fondamentaux.

II.3.2. Caractéristiques particulières des fonctions de mise au point.

Avant d'exposer une implémentation particulière des fonctions précédemment décrites, donnons un aperçu et une justification des caractéristiques que nous avons voulu attacher à ces fonctions.

a) Aucune instruction supplémentaire n'a été ajoutée dans le code objet issu de la compilation d'un module GSL, ni dans le texte source. Le code objet est ainsi strictement identique en contenu et en volume avec ou sans les fonctions de mise au point.

On peut apporter plusieurs justifications à cette caractéristique :

- On se trouve dans un contexte de programmation système ; or, très souvent, la place occupée par un module est un facteur important. De plus, le contrôle de la place étant une caractéristique de génération du compilateur GSL, on a essayé de la conserver.
- Les fonctions de mise au point doivent être disponibles à tout moment, sans nécessiter une recompilation des modules ni régénération du système en construction (même en exploitation normale).
- On n'accroît pas le temps d'exécution du système en construction par l'exécution de code supplémentaire lorsque cela n'est pas nécessaire.
- En ne modifiant pas le code objet, on est assuré d'avoir une compatibilité totale entre des modules au point et ceux en construction.

b) Les fonctions présentées travaillent de manière interactive avec l'utilisateur. Ce dernier peut ainsi prendre les décisions au cours de l'exécution de son système et non à priori comme dans un système "batch". De plus, l'implémentation de ce système a été faite dans le contexte du système conversationnel CMS [I1] ; il était donc impensable de ne pas profiter de cette possibilité.

c) On a voulu donner à ces fonctions une grande fiabilité, en faisant parfois des vérifications qui peuvent paraître inutiles (Vérification de valeurs de pointeurs,de bornes de tableaux, de vie d'une variable, etc...).

Il est vrai que le cumul des coûts (temps et informations nécessaires) de ces vérifications peut paraître élevé. Cependant une erreur lors d'une mise au point nécessite parfois beaucoup de travail pour se replacer dans le même contexte.

De surcroit, il est quelque peu aberrant de ne pas profiter de la structuration des données et du programme que fournit l'utilisation d'un langage de haut niveau et de ne réaliser qu'une mise au point symbolique, guère plus évoluée que celle de programmes écrits en langage machine.

d) La protection des fonctions d'aide à la mise au point, vis-à-vis des erreurs du système en construction, est nécessaire si l'on veut disposer d'un outil apte à fonctionner quelles que soient les erreurs. Elle est totale lors du fonctionnement dans le cadre du système SPY [L3].

CHAPITRE III

PRESENTATION D' UNE IMPLEMENTATION

Le but de ce chapitre est de décrire les traits principaux d'une implémentation des outils proposés ; ainsi, nous avons volontairement laissé de côté de fastidieux détails techniques n'apportant pas une compréhension meilleure du système proposé.

III-1 - RAPPEL DU CADRE DE FONCTIONNEMENT

Les outils proposés nécessitent un support ; ce dernier doit remplir deux fonctions :

* La Gestion des fichiers du système proposé.
* La Gestion des interruptions.

Le système proposé fonctionne indifféremment sur une machine virtuelle ou dans le cadre d'une machine "SPY". [B7].

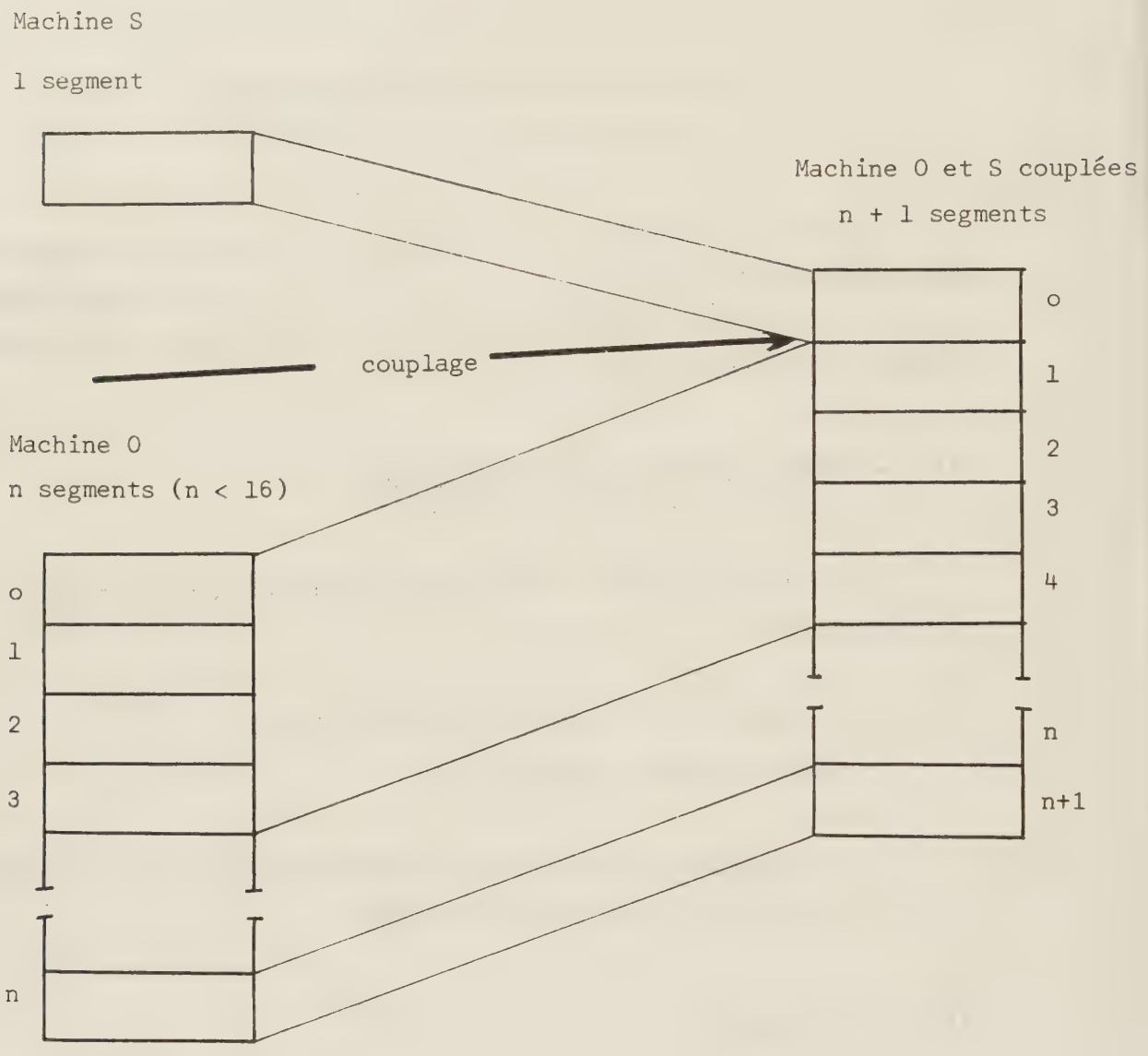
III-1-1 - Système SPY

Il s'agit d'un couplage, réalisé par l'intermédiaire des mécanismes de CP [I8], de deux machines virtuelles (cf. schéma 1).

Le couplage peut être conditionné par des événements tels que demande d'exécution d'instructions privilégiées, ou arrivée d'une interruption machine.

Supposons que l'on dispose de deux machines virtuelles S (SPY) et O (OBJET), chacune possédant des ressources propres. Lorsque le couplage est

Schéma 1 : Couplage au niveau des mémoires centrales.



Les segments 0, 1, 2, 3...n de la machine 0 deviennent les segments 1, 2, 3, 4...n+1 de la machine couplée 0.S.

Les adresses des éléments appartenant à la machine 0 subissent donc une translation dont la valeur est égale à la taille de la mémoire de la machine S, lorsque le couplage est établi.

établi, un système s'exécutant dans S peut observer l'état de la mémoire de O ; lorsque le système appartenant à O s'exécute, il ne peut atteindre S.

Une application de ce couplage est l'aide à la mise au point. (En particulier, celle de système destiné à s'exécuter sur une machine nue).

La machine O contient le système à tester ou système OBJET, la machine S contient un système d'aide à la mise au point.

Les outils proposés peuvent utiliser une telle possibilité ; le système OBJET réside alors dans la machine O, les outils dans le segment de la machine S.

- Organisation de la machine S -

La description de la mémoire de la machine virtuelle S est fournie par la figure 2 ; elle contient :

* Noyau de CMS modifié.
* Système proposé (d'intégration et de mise au point).
* Module gestion des interruptions.
* Système d'aide à la mise au point niveau langage machine.

a) La gestion des fichiers nécessairesau fonctionnement des outils est assurée par le noyau de CMS [Il] modifié.

Ce dernier s'occupe d'exécuter les actions demandées, soit :

- Créer un fichier.
- Supprimer un fichier.
- Lire des enregistrements d'un fichier.
- Réécrire des enregistrements.

  etc....

Il incombe ainsi à ce noyau de CMS d'assumer la gestion de l'espace disque de la machine S.

b) Une autre fonction prise en charge par le noyau CMS a trait aux entrées-sorties sur les périphériques tels que :

- Imprimante.
- Télétype.
- Bandes.

  etc...

De cette manière, toute demande d'entrée/sortie émise par l'outil est faite à un niveau élevé, sans aucun souci relatif aux mécanismes sous-jacents.

c) La gestion des interruptions est assurée par un module spécial. Ce point est très important ; en effet, le contrôle de l'exécution est réalisé par l'intermédiaire d'interruptions programme.

d) Le noyau CMS assume aussi la responsabilité de la gestion de la mémoire libre, satisfait les demandes des divers composants de l'outil proposé et récupère la place libérée.

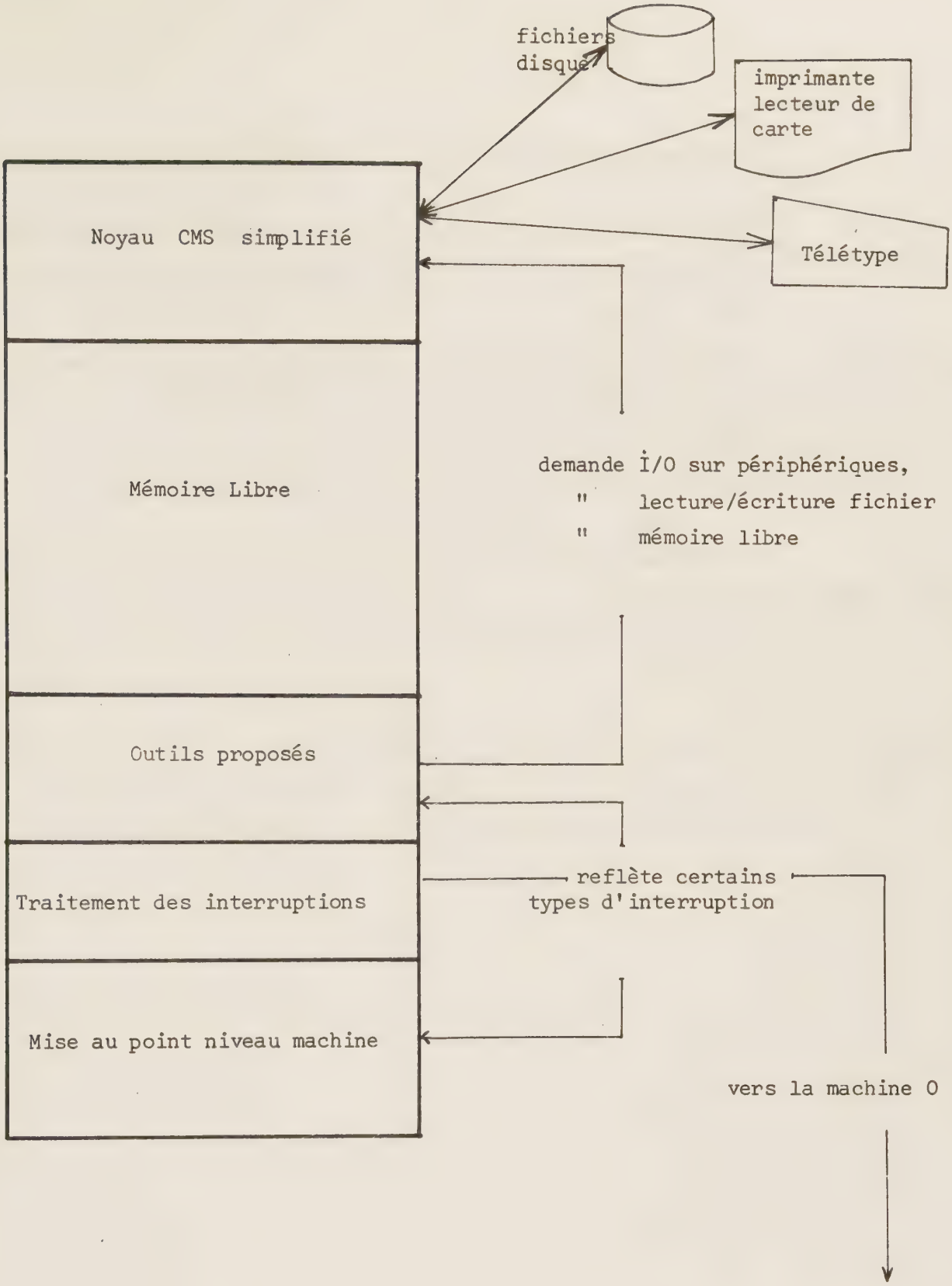


Fig. 2 - Organisation de la mémoire de la machine virtuelle S.

Les principaux avantages de l'utilisation d'un tel support sont :

* Eviter d'interférer avec le système OBJET lors de la mise au point, au niveau de la gestion des différentes ressources (périphériques, mémoire, etc...).

* Protéger le système de mise au point. Lorsque le système résidant dans la machine O s'exécute, il lui est impossible d'adresser la mémoire de la machine S.

* Permettre la mise au point de systèmes d'exploitation.

Cependant, la mise en oeuvre d'un tel système peut, dans certains cas, s'avérer lourde.

III-1-2 - Machine unique

Dans ce cas, l'outil, ainsi que le système OBJET, cohabitent dans le même espace mémoire (figure 3).

Le système CMS assume, dans ce cas précis, la gestion des ressources à la fois pour le compte du système OBJET et pour celui de l'outil (gestion de fichiers, périphérique, etc...). La seule exception qui soit faite est relative à la gestion de la mémoire libre, l'outil possédant sa propre mémoire libre ainsi qu'un composant destiné à la gérer.

Le principal reproche que l'on peut faire à une telle organisation est de n'assurer aucune protection efficace de l'outil d'aide à la mise au point durant la phase de test.

De plus, le noyau de CMS doit subir une légère modification, afin d'être capable de refléter certains types d'interruption vers l'outil de mise au point assistée, pendant l'exécution du système OBJET.

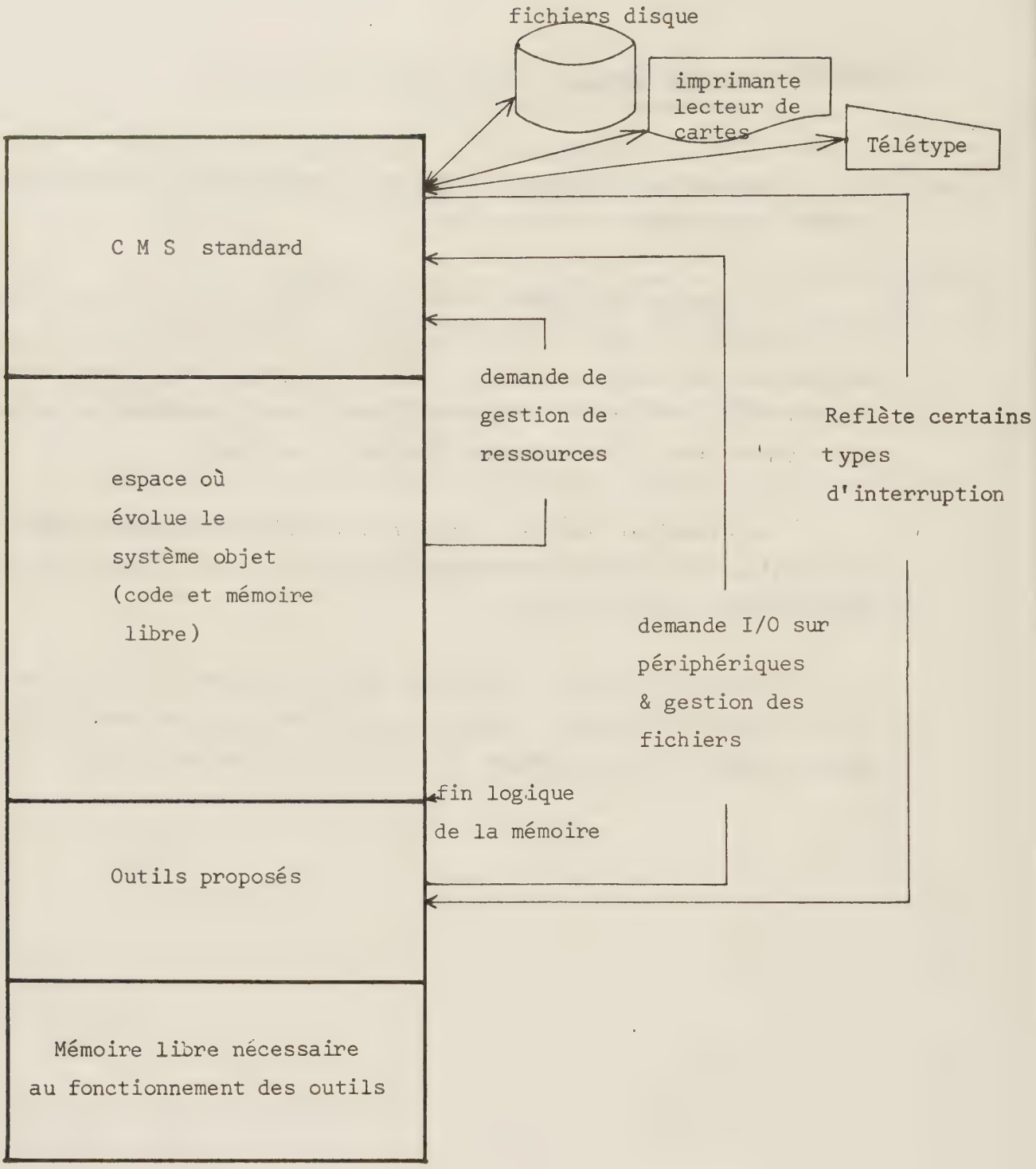


Fig. 3 - Cas d'une machine unique.

III - 2 - INFORMATION FOURNIE PAR LE COMPILATEUR

Le fonctionnement des outils proposés nécessite la connaissance des modules composant le système en construction. L'obtention de l'information décrivant ces modules a lieu au cours de la phase de compilation de chacun d'eux.

a) L'information -

a-1) La description des objets.

Elle permet de connaître le nom et les caractéristiques des variables manipulées dans un module.

On remarquera qu'à l'intérieur d'un même programme GSL, il y a correspondance "injective" entre le nom (identificateur) et la variable, contrairement à des langages possédant une structure de bloc tel qu'ALGOL. Dans certains cas seulement, en GSL, la variable peut ne pas avoir d'existence (variable basée ou de type registre). Pour cette raison, nous confondons parfois la variable avec l'identificateur qui la repère.

a-2) La description des opérations.

Elle permet de connaître les numéros de ligne où sont déclarés, utilisés, modifiés, les identificateurs apparus dans le module.

a-3) La correspondance entre le code source et le code objet.

Cette information permet d'établir la correspondance entre le numéro (affecté par le compilateur) de l'instruction GSL et l'adresse relative, par rapport au début du module de la première instruction machine générée.

Exemple : Soit l'instruction 25 d'un programme GSL. Nous donnons l'instruction GSL et le code objet engendré.

```
DØ    R7 = 1 TO 6        } 25ème instruction GSL d'un programme
      LA    R7,0001   ←  déplacement 54₁₆ } code objet
@E6   DS    OH                            } engendré
      C     R7, @DC00 ←  déplacement 58₁₆ }
```

On aura, dans la table, la correspondance du numéro 25 d'instruction GSL et du déplacement 54_{16}. (@DC00 est le nom symbolique de la variable générée par le compilateur et contenant 6).

a-4) La description des registres.

Elle permet de connaître, pour chaque instruction GSL, l'ensemble des registres significatifs. On a trois classes de registres significatifs :

- Les registres invariants. Ce sont ceux bloqués dans l'environnement ou servant à l'adressage, par exemple.
- Les registres alloués et gérés par l'utilisateur.
- Les registres gérés par le compilateur et utilisés dans l'optimisation.

a-5) Description des utilisations de variables indicées.

Elle permet d'avoir une description des points d'utilisation des variables indicées.

b) Organisation des informations gardées à la compilation -

b-1) Modification du compilateur.

Des modifications ont été apportées au compilateur GSL, de manière à collecter ces renseignements dès leur création. Toutefois, ceci n'est pas systématique et n'a lieu que lorsque la compilation est demandée avec l'option "DEBUG".

Ces informations sont placées dans un fichier descripteur du code OBJET engendré. Son nom est identique à celui du fichier SOURCE GSL dont il est issu ; son type est "DEBUG". Le système CMS assure la gestion de ces fichiers.

Exemple : L'utilisateur veut compiler le module M1 contenu dans le fichier CMS de même nom. Il désire obtenir une liste de la compilation, et par la suite, utiliser les outils proposés. Il émet la commande :

GSL M1 (PRINT DEBUG)

Les opérations effectuées peuvent être représentées par le schéma 4 :

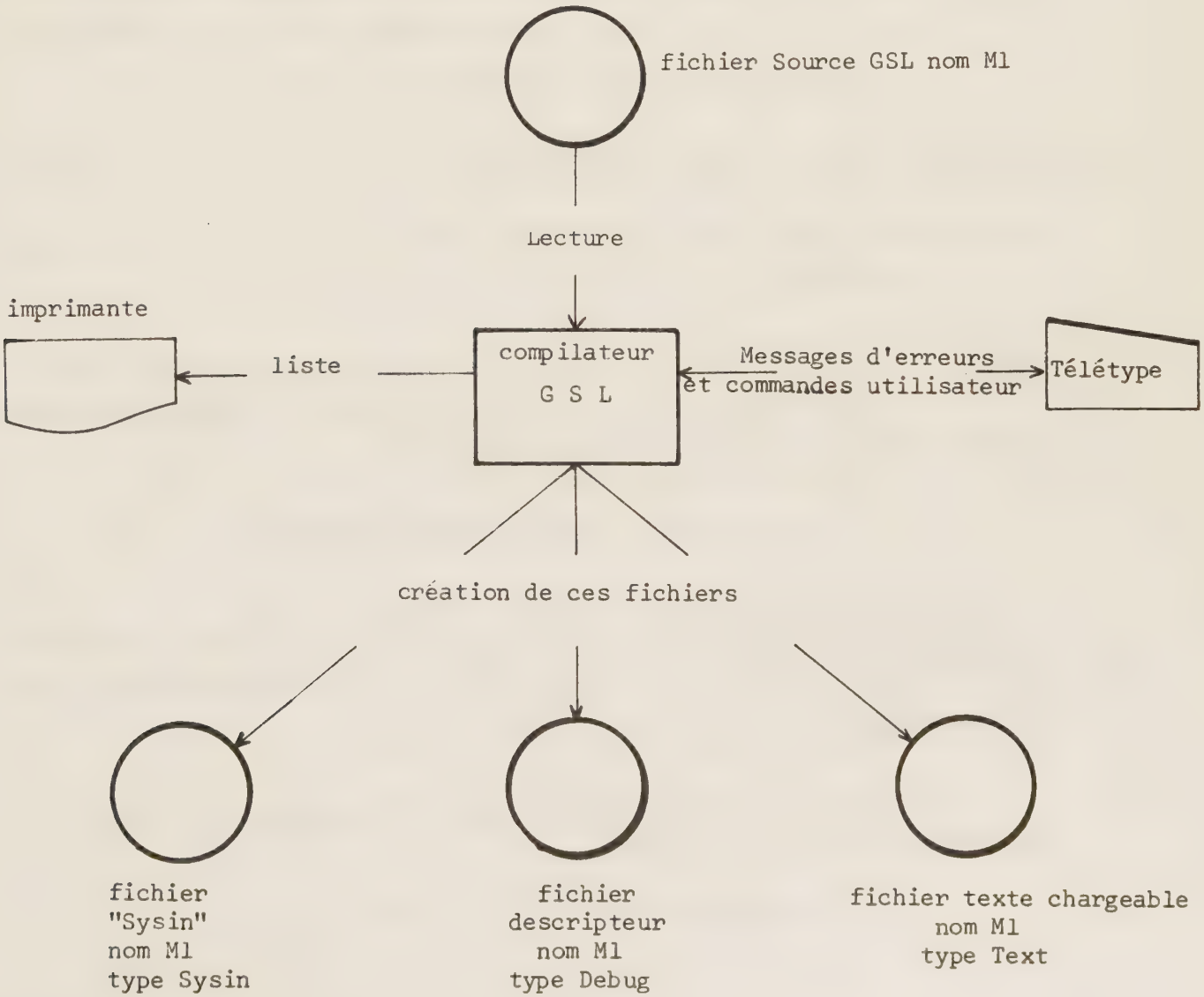


Schéma 4.

b-2) Structuration du fichier "DEBUG" engendré par le compilateur.

b-2-1) Division de l'information.

L'ensemble des informations conservées à la compilation ou engendrées au cours de la mise au point a été divisé en groupes ; la structure adoptée a eu pour but de :

* Diviser logiquement: c'est-à-dire tous les éléments appartenant à un groupe sont de même nature ou répondent à la même relation.
  Exemple : descripteur des variables, descripteur d'utilisation.

* Regrouper (dans un même module) les éléments invariants, d'une part, et les éléments susceptibles d'être modifiés entre deux compilations, ou au cours d'une session, d'autre part.

* Permettre une recherche aisée d'un groupe et d'un élément dans un groupe.

* Trouver un compromis entre la place occupée sur disque et celle occupée en mémoire.

Remarque : Tout adressage à l'intérieur d'un groupe est réalisé relativement à une base ; ceci permet de translater aisément les éléments d'un groupe ou le groupe tout entier.

b-2-2) Structuration globale du fichier.

La structure du fichier permet d'accéder directement à un groupe et d'en connaître, avant la lecture, toutes les caractéristiques qui lui sont attachées.

Les enregistrements sont de longueur fixe, le premier servant à décrire les groupes qui suivent.

Pour chaque groupe, on trouve dans ce descripteur :

* La taille en octets du groupe.
* Le numéro du premier enregistrement où débute le groupe dans le fichier.
* Le numéro du dernier enregistrement où se termine le groupe.
* La taille du groupe en nombre d'enregistrements.

Ainsi le processus de lecture du groupe i est le suivant :

* Lecture du premier enregistrement (descripteur).
* Obtention de la place mémoire nécessaire au groupe i et mise à jour de sa description.
* Lecture en bloc de tout le groupe i intéressé.

b-2-3) Les groupes et l'organisation de chacun d'eux.

b-2-3-1) La table des identificateurs.

Elle se compose d'une table de "HASHCODE" de 128 entrées et d'un tas contenant les descripteurs des variables, chaînés entre eux lorsque leur nom donne le même résultat de "HASHCODE". Le premier descripteur est chaîné à partir de cette table (le résultat de la fonction de "HASHCODE" sert d'index dans cette table).

Soit : - id le nom ou l'identificateur de la variable.
- card (id) le nombre de caractères non blancs composant l'identificateur.
- id (i) le $i^{ème}$ caractère de l'identificateur.

La fonction de "HASHCODE" f (id) est la suivante :

$$f(id) = \left(\sum_{i=1}^{card(id)} id(i) \right) \text{ modulo } 128.$$

b-2-3-2) La table des références croisées.

Elle a la même structure que la table des identificateurs:table de "HASHCODE" et tas des descripteurs (la fonction de "HASHCODE" est la même).

Chaque descripteur est relatif à un identificateur et donne :

* Le déplacement du descripteur de la variable dans la table des identificateurs.
* Les numéros de ligne où cette variable est utilisée et le type de l'utilisation.

b-2-3-3) Table de correspondance.

La forme d'une entrée dans cette table est la suivante :

(1)	(2)	(3)

(1) Numéro de l'instruction GSL source.

(2) Déplacement de la première instruction assembleur engendrée.

(3) Pointeur (relatif) vers le descripteur des registres significatifs associé à l'instruction.

Elle permet d'associer en plus, à l'instruction, l'ensemble des registres significatifs (alloués par l'utilisateur, ou bloqués, ou servant à l'optimisation).

Les recherches, dans cette table, se font sur deux critères :

- Soit sur comparaison de numéros d'instruction GSL, c'est-à-dire sur le champ (1).
- Soit sur comparaison de déplacements d'instruction machine, c'est-à-dire sur le champ (2).

On remarque que cette table est ordonnée suivant chacun des champs (1) ou (2).

Exemple : Soit deux entrées fournies par deux index i et j.
i(1) étant le champ (1) de l'entrée i et j(1) celui de l'entrée j

$$i > j \iff i(1) > j(1)$$

$$i > j \iff i(2) > j(2)$$

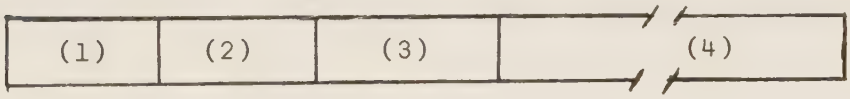
Vu cette propriété, nous avons pensé à utiliser une méthode de recherche par dichotomie. Cependant, dans une mémoire paginée, cette méthode peut s'avérer coûteuse.

b-2-3-4) Organisation des descripteurs des registres.

L'information décrivant les registres est structurée de manière à minimiser la place occupée sur disque ; on trouve :

* En tête, les registres invariants tout au long du module.
* Une série de descripteurs associés à chaque instruction.

Chaque descripteur a la forme suivante :



(1) indique l'ensemble des registres significatifs (1 bit par registre).

(2) indique (1 bit par registre),pour chaque registre significatif, si sa description est dans la partie (4) ou est restée invariante par rapport à l'instruction précédente.

(3) Pointeur (relatif) vers le descripteur des registres attaché à l'instruction précédente.

(4) Descripteurs des registres significatifs et modifiés par rapport à l'instruction précédente.

Pour une instruction donnée, la recherche d'un registre se fait de la manière suivante :

. On vérifie si le registre est significatif. S'il ne l'est pas, on le recherche dans les registres invariants.

. S'il est significatif, et si son descripteur est inchangé par rapport à l'instruction précédente, on recommence la recherche au niveau du descripteur des registres attaché à l'instruction précédente.

. Dès que le descripteur du registre est présent dans le champ (4), la recherche a abouti.

b-2-3-5) Tables de contrôle

Leur structure est donnée dans le paragraphe décrivant le fonctionnement de l'outil d'aide à la mise au point.

III - 3 - CHARGEMENT ASSISTE

Comme nous l'avons vu dans le chapitre précédent, nous avons deux fonctions, une fonction de vérification et une fonction de chargement.

Cette dernière peut prendre plusieurs aspects suivant le cadre dans lequel on travaille :

* Soit simuler le travail d'un chargeur d'"IPL" ("initial program loading", consiste à amener dans la mémoire nue le noyau d'un système pour permettre l'exploitation du calculateur). Dans ce cas, on utilise le support SPY, la machine nue étant la machine où va s'exécuter le système OBJET. Cet aspect est extrêmement utile lorsqu'on travaille à la construction d'un système d'exploitation.

* Soit exécuter le travail d'un chargeur standard, lorsque l'on opère dans une seule machine.

Dans tous les cas, les possibilités offertes sont identiques ; le fonctionnement du chargeur assisté reste inchangé.

Toutefois, il a été conçu en vue d'offrir des possibilités plus étendues que celles disponibles dans un chargeur standard [Ill0], lorsque les modules à traiter proviennent de la compilation de programmes GSL.

III-3-1 - Spécifications externes

On retrouve l'ensemble des possibilités disponibles dans le cadre d'un chargeur standard (chargeur absolu, chargeur translateur, résolution des adresses externes). Le code machine à charger est issu d'origines diverses : assemblage, compilation, etc...

Lorsque les modules objets à charger proviennent de compilation GSL, de nouvelles possibilités sont offertes.

a) Au niveau de la fonction de chargement -

a-1) Séparation du code et des données

Cette possibilité est très intéressante lors de la création de composants devant être partagés sur un calculateur possédant un mécanisme de pagination.

Lors de la génération du code machine à partir du code source, le compilateur GSL est capable de séparer les éléments invariants du programme (code et constantes) des éléments variables ou susceptibles d'être modifiés au cours de l'exécution.

Le chargeur assisté, au cours de son fonctionnement, est capable de différencier ces deux constituants et de les placer dans différentes parties de la mémoire suivant les ordres spécifiés par l'utilisateur.

Si l'on dispose d'un mécanisme de segmentation tel que celui disponible sous CP-67 [18] par exemple, plusieurs processus peuvent se partager la même zone de code exécutable, chacun d'eux possédant sa propre zone de données (le partage se faisant au niveau du mécanisme d'adressage).

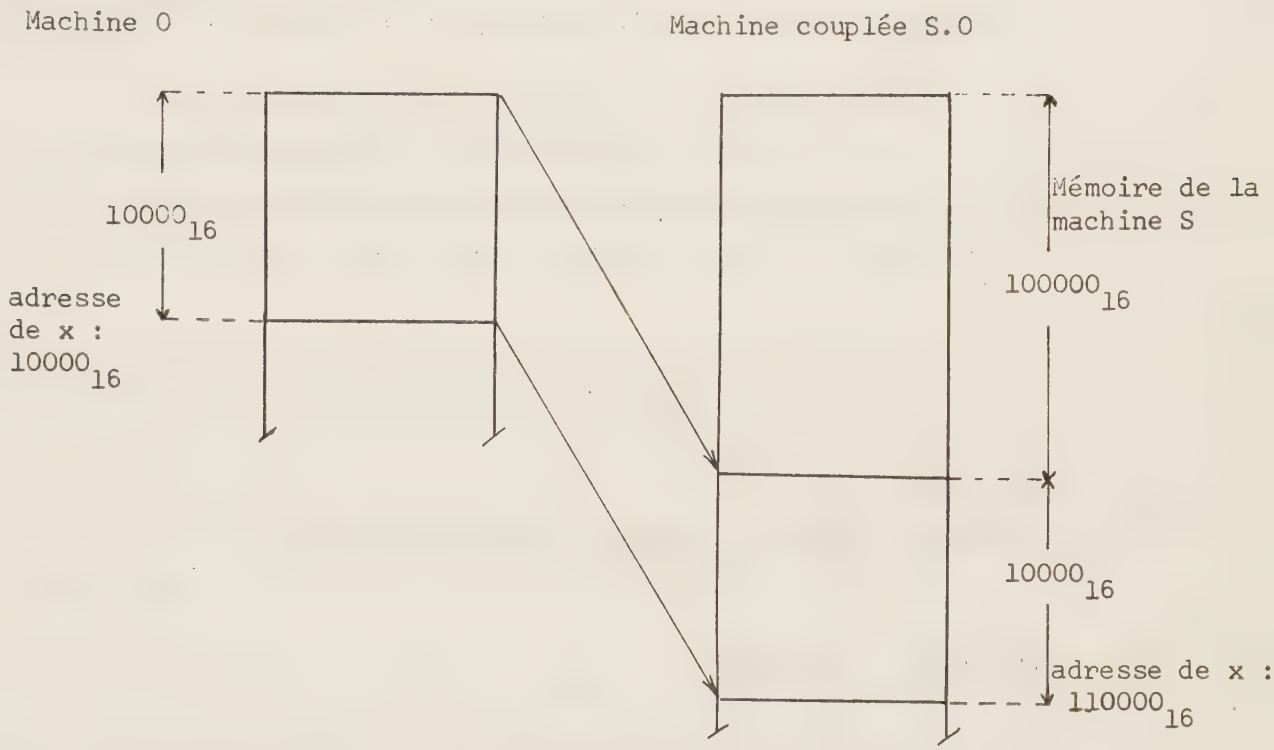
a-2) Translation du code et des données

En premier lieu, lorsque le support est le système SPY, il existe une translation implicite.

En effet, les segments 0 1 r...n (n < 15) de la machine 0 (cf. exemple, paragraphe III-1) deviennent les segments 1 2 ... n+1 de la mémoire de la machine couplée, le premier segment étant celui de la machine $\mathcal{S}$. Lorsque l'on charge à partir de la machine $\mathcal{S}$, il faut effectuer une translation dont la valeur est égale à la taille de $\mathcal{S}$ (elle est réalisée automatiquement).

Exemple : Supposons que X soit à l'adresse '10 000' hexadimal dans la mémoire de la machine 0 ; lorsque le couplage est effectué, il est à l'adresse

"110 000" hexdécimal de la machine couplée.



L'utilisateur peut, s'il le désire, effectuer le chargement du code ou des données dans un segment et spécifier que l'exécution aura lieu dans un autre segment.

Ce problème se ramène à une translation d'adresse. Cette possibilité est utilisée lorsque le segment d'exécution est occupé ou n'existe pas sur la machine dont on dispose.

a-3) Constitution de groupes translatables

Une possibilité non encore exploitée est la création de groupes translatables de modules avec une description complète des liens qui les unissent. Ceci peut être utile lors de la construction de système important, où l'on éviterait ainsi le chargement d'une partie n'ayant pas été modifiée.

b) Au niveau de la fonction de vérification -

Lors du chargement de modules objet issus de compilation GSL et possédant l'option "DEBUG", le chargeur est capable de vérifier la cohérence des liaisons.

Cette fonction est une des plus originales et intéressantes puisqu'elle permet, lors de l'intégration des modules du système OBJET, de détecter des erreurs dont la présence, sans cela, n'aurait été révélée qu'à l'exécution.

Le travail de vérification se fait par l'exploitation de l'information contenue dans les fichiers de type "DEBUG".

III-3-2 - Commandes, cartes de contrôle, erreurs

III-3-2-1 - Commandes

On dispose de deux commandes pour activer le chargement assisté.

a) La commande "load" permet d'amener en mémoire un premier ensemble de modules et de résoudre les liens qui les unissent.

Sa forme est la suivante :

LOAD <NOM> <OPTION>

<NOM> représente un nom de fichier "EXEC" de CMS donnant la liste des éléments à charger.

<OPTION> indique s'il doit y avoir création ou non de fichiers spéciaux, ou d'impression de listes à la console, etc ...

b) La commande "use" permet de rappeler le chargeur après l'exécution de la commande "LOAD", afin de poursuivre le chargement du système en construction. La forme de la commande est :

USE <NOMS> <OPTIONS>.

Les options sont identiques à celles de la commande "LOAD", le premier paramètre représente une liste de noms de modules à charger.

III-3-2-2 - Cartes de contrôle

On retrouvera l'ensemble des cartes admises par un chargeur classique [I10].

Pour permettre le contrôle supplémentaire précédemment décrit, de nouvelles cartes ont été ajoutées ; ce sont :

a) La carte "COD" permet de spécifier les emplacements de chargement et d'exécution du code du système à charger ; sa forme est la suivante :
. COD <SG1> <SG2>.

SG1 indique le numéro de segment où aura lieu l'exécution
SG2 indique le numéro de segment où aura lieu le chargement

b) La carte "DAT" permet de spécifier les emplacements de chargement et d'exécution des données du système à charger. Sa forme syntaxique et la signification des paramètres sont identiques à la carte "COD".

c) La carte "SLA" permet de modifier l'adresse de chargement des modules qui suivent. Sa forme est : . SLA <SG> <ADR>.

Elle indique que le chargement doit se poursuivre dans le segment numéro <SG> , à l'adresse spécifiée par <ADR> ; le segment spécifié doit être un segment de code.

d) La carte"SPB"permet de forcer le chargement d'un module sur une frontière de page ou de demi-page. Sa forme est : SPB <2K| 4K>

III-3-2-3 - Erreurs

Durant la phase de chargement assisté, on s'efforce de détecter le maximum d'erreurs. Elles peuvent se classer en plusieurs types :

a) Les erreurs de chargement -

a.1) Les erreurs classiques détectées par un chargeur : débordement de tables, symboles non résolus, etc ...

a.2) Des erreurs dues à des manipulations erronées des commandes "LOAD" et "USE", ainsi que des cartes de contrôle. On évite, dans ces conditions, de détruire le code déjà chargé ou d'être en désaccord avec la configuration de la mémoire virtuelle de la machine dont on dispose.

b) Les erreurs de liaison entre modules -

Le troisième type d'erreurs a trait à des incohérences de liens au niveau du texte source GSL ; on peut les qualifier d'erreurs d'intégration. Elles sont séparées en deux classes suivant leur gravité :

b.1) La classe des avertissements regroupe les erreurs qui peuvent ne pas être fatales durant la phase d'exécution.

b.2) La classe des erreurs sévères ; l'erreur détectée a de grandes chances d'entraver une exécution correcte du système OBJET.

Donnons des exemples de ces différents types d'erreurs :

Exemple 1 -

```
COD 01      TEXT  }  Cartes de
                  }  contrôle forme : . COD   0   1
DAT 12      TEXT  }                   . DAT   1   2

ALPHA       TEXT  }  fichier  ALPHA  à charger
SLA01000    TEXT  }  format   SLA    0   10000
```

BETA TEXT } fichier BETA chargé en 10 000 du segment 0
SPB4K TEXT } format interne .SPB 0 4K
GAMMA TEXT } fichier GAMMA chargé sur la prochaine frontière de page.

Supposons que l'on remplace SLA010000 par une carte de la forme SLA 2 10 000 : une erreur serait détectée puisque le segment 2 n'a pas été alloué pour le chargement.

Exemple 2 -

Supposons que nous chargeons deux fichiers ALPHA et BETA issus de fichiers source GSL et ayant été compilés avec l'option "DEBUG".

ALPHA contient la déclaration :

```
DCL   ZZZ   BYTE (8)   ALD (8,0)   EXT ;
```

BETA contient la déclaration :

```
DCL   ZZZ   BYTE (8)   ALD (2,0)   ENTRY ;
```

L'attribut "ALD" signifie que la variable est alignée sur une frontière de demi mot pour (2,0) et de double mot pour (8,0).

Lors du chargement, une erreur sévère sera détectée, la contrainte (8,0) étant plus forte que la contrainte (2,0).

Supposons, par contre, que l'on ait les déclarations :

ALPHA contient :

```
DCL   ZZZ   BYTE (8)   ALD (2,0)   EXT ;
```

BETA contient :

```
DCL   ZZZ   BYTE (8)   ALD (8,0)   ENTRY ;
```

Lors du chargement, il n'y aura qu'un avertissement, la contrainte (8,0) de l'"ENTRY" étant plus forte que celle de l'"EXTERN".

Exemple 3 -

Soit les deux déclarations de la même procédure où les paramètres sont erronés :

L'en-tête de la procédure P :

```
P : PROCEDURE ( R1 REG(1) FIXED LONG, R2 REG(2) POINTER) ;
```

Et la déclaration, dans un autre module de la même procédure :

```
DCL  P  PROCEDURE ( REG(1) POINTER, REG(2) FIXED LONG) EXT ;
```

La fonction de vérification signalera au programmeur une erreur appartenant à la classe des erreurs sévères, au niveau de la correspondance des paramètres.

III-3-3 - Logique interne et résultats annexes

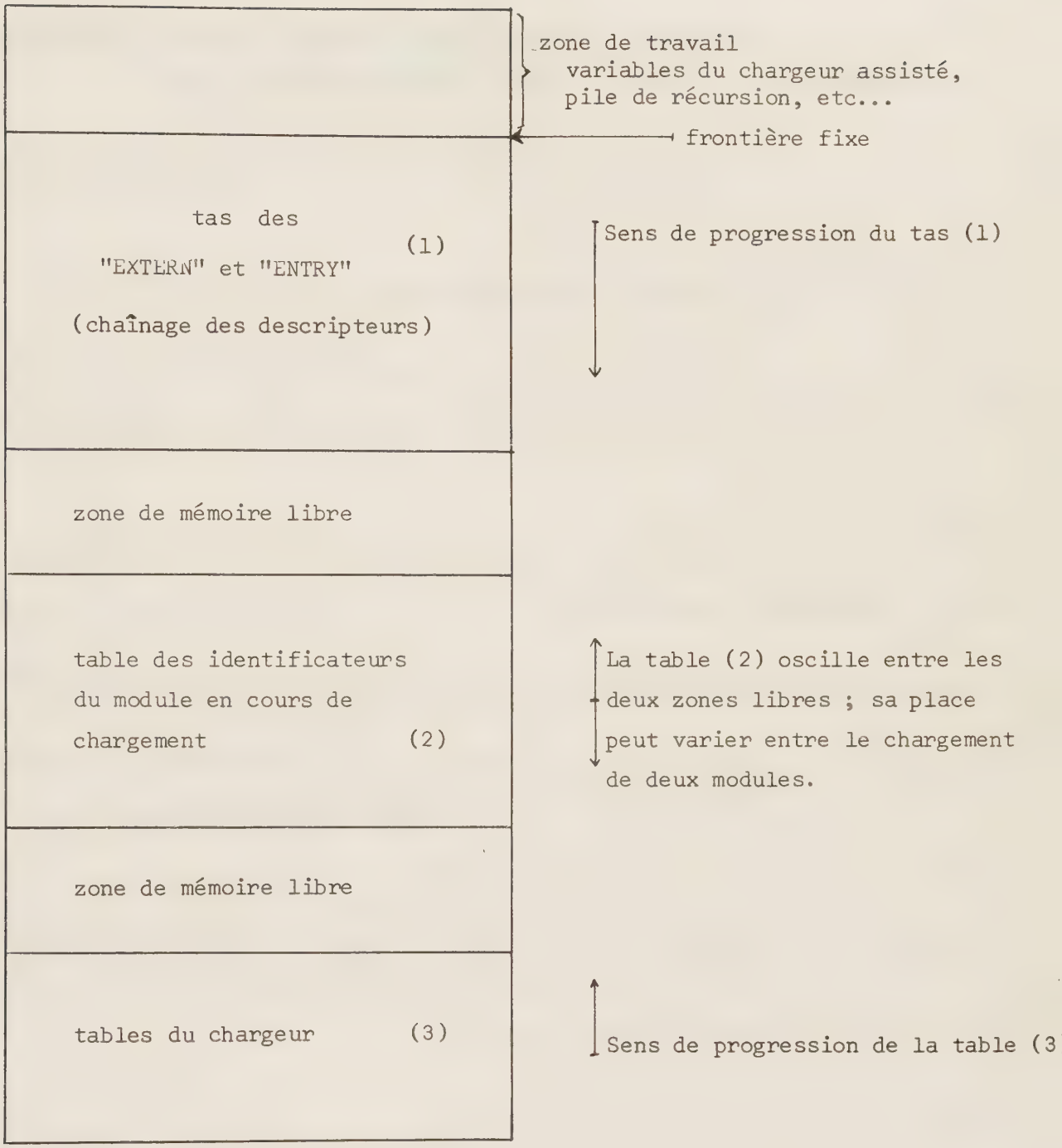
Dans ce paragraphe, nous nous intéressons tout particulièrement à la partie des vérifications du chargeur assisté.

a) Initialisation -

Cette phase consiste à organiser la mémoire et à prendre connaissance de la machine sur laquelle on va faire fonctionner le système OBJET.

a.1) On calcule la taille de chaque segment, on "trace une topographie" de la mémoire (ceci pour tenir compte des segments incomplets).

a.2) On acquiert de la place pour les données et les tables ; on répartit cette place entre ses diverses affectations.



. Lorsque la taille des deux zones libres entre deux chargements est bien choisie, aucun déplacement de la table (2) n'est nécessaire.

Figure 5 - Configuration de la mémoire utilisée par le chargeur assisté.

Cette partie est entièrement paramétrée afin de permettre des changements aisés lorsque l'on passe d'un système à un autre.

La répartition de l'espace mémoire est décrite par la figure 5 ; sa gestion est assurée dynamiquement par une fonction spéciale du chargeur assisté

b) Phase de collecte des descripteurs

A ce niveau on lit, pour chaque module, le fichier contenant le code objet (il est traité par la fonction de chargement) et le fichier de type "DEBUG" attaché au module ; ce fichier est traité par la fonction de vérification.

Chaque élément de liaison donne naissance à une entrée dans une table (table du chargeur).

A chacune de ces entrées sont associées deux listes de descripteurs :

- Les descripteurs des points où doit être placée l'adresse de l'élément (ceci intéresse la fonction de chargement).

- Les descriptions au niveau GSL fournies par les diverses déclarations de l'élément dans les différents modules composant le système en construction (ces descripteurs sont fournis par la table des identificateurs attachée au module et conservée dans le fichier de type "DEBUG" correspondant).

Exemple - Supposons que l'élément F apparaisse dans les modules A B et C comme "EXTERN" et dans D comme "ENTRY" ; on aura la configuration donnée par la figure 6.

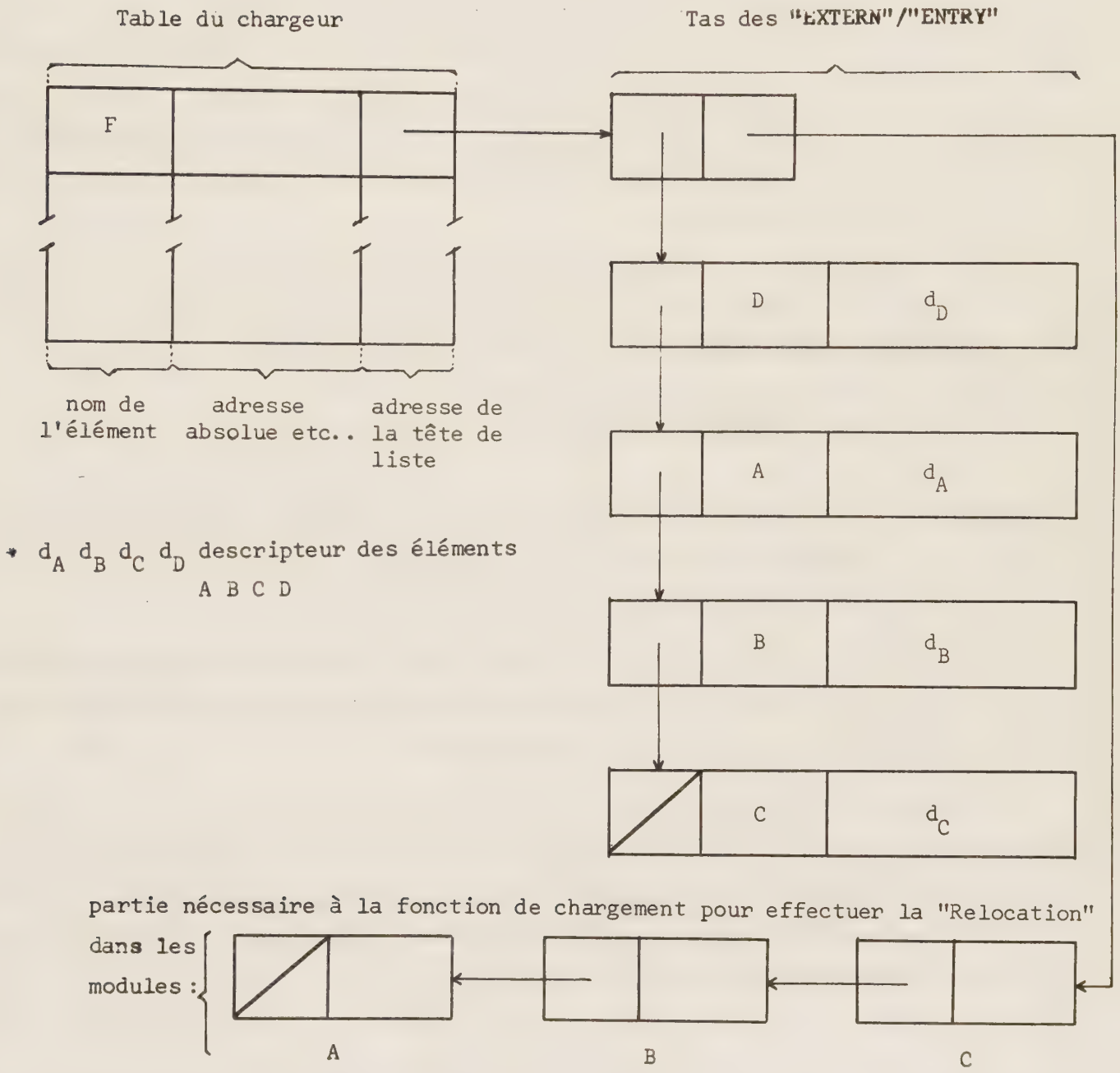


Figure 6 : chaînage des descripteurs.

c) Vérification de cohérence d'utilisation

Lorsque tous les modules ont été chargés, on peut procéder à :

- La mise en place des constantes adresses (bien que cette fonction puisse aussi être effectuée au cours du chargement).
- La vérification de la cohérence des utilisations. Ceci consiste en la comparaison des différents descripteurs des "EXTERN" avec le descripteur de l'"ENTRY", chaînés à partir de la même entrée de la table du chargeur (le descripteur de l'"ENTRY" est toujours en tête de liste).

Le descripteur d'un élément peut lui-même posséder une structure arborescente, par exemple dans le cas d'une procédure.

d) Résultats annexes

Au cours de son fonctionnement, le chargeur assisté produit une table spéciale qui porte le nom de "DIRECTORY" et dont l'utilisation apparaît au cours de la phase de mise au point assistée.

Si l'on observe la figure 2, on s'aperçoit qu'en fin de chargement on possède la matrice des appels statiques des procédures et de l'utilisation des variables globales du système OBJET.

Ainsi, reprenons l'exemple de la figure 6 ; à partir des descripteurs, on peut reconstituer la déclaration. On a, alors, l'interprétation suivante: l'identificateur F est déclaré et appartient au module D avec la déclaration d_D ; il est utilisé dans les modules A B C avec, respectivement, les descripteurs d_A d_B et d_C;

Des renseignements complémentaires plus fins peuvent être obtenus avec l'exploitation de la table des références croisées, engendrée lors de la compilation des modules A, B, C et D.

III - 4 - MISE AU POINT ASSISTEE

III-4-1 - Idées générales

Au cours du chapitre précédent, nous avons vu des aspects généraux ; nous nous attachons, ici, à donner des idées relatives à une implémentation.

a) Lors de l'émission de commandes par l'utilisateur ou lors de contrôles internes provenant du système de mise au point assistée, des tables seront nécessaires.

La politique suivante a été adoptée :

- On ne lit une table que si sa présence en mémoire est absolument nécessaire, sans anticiper sur ce que va faire l'utilisateur.

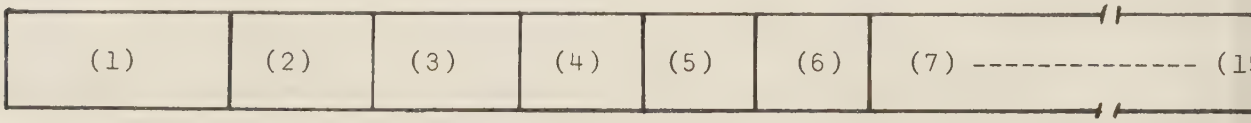
- On reprend en priorité la place occupée par des tables ne nécessitant pas de réécriture sur disque.

Ce deuxième choix peut être affiné en affectant à chaque table un temps de résidence, par exemple, ou en éliminant celles appartenant à des modules non actifs. Toutefois, ces améliorations s'avèrent vite coûteuses.

b) Le fonctionnement de la mise au point assistée repose sur la connaissance qu'elle possède du système à mettre au point (ou système OBJET). Toutes les informations nécessaires seront accessibles à partir d'une table principale ou "DIRECTORY". Cette table est en partie constituée lors de la phase de chargement assisté ; elle possède une entrée par module composant du système OBJET (cf. figure 7).

Pendant l'exécution assistée, chaque entrée permet d'accéder à l'information caractérisant le module, directement en fournissant une base si l'information est en mémoire, indirectement dans le cas contraire, en fournissant le nom du fichier la contenant.

Chaque entrée a une longueur variable ; en effet, plusieurs noms peuvent être rattachés au même module ; seul le nom du fichier contenant le texte source GSL est unique. Toute recherche, dans cette table, est séquentielle.



Information fournie par le compilateur et le chargeur, figée au cours de la session de mise au point :

(1) Nom du fichier où se trouve l'information décrivant le module.

(2) Adresse d'implantation du module en mémoire.

(3) Taille occupée par le code objet du module.

(5) Nombre de noms attachés à la procédure principale.

(6) Adresse de la zone des données du module.

(15) Suite des noms attachés à la procédure principale.

Information variable au cours de la session de mise au point.

(4) Indicateur (d'arrêt de macro-instruction, etc...).

(7) Adresse de la table des identificateurs.

(8) " de la table des références croisées.

(9) " de la table de contrôle.

(10) " de la table de correspondance.

(11) " de l'ensemble des descripteurs des registres.

(12) " du vecteur des arrêts sur référence.

(13) " de la table de contrôle d'utilisation des variables indicées.

(14) " de l'ensemble des descripteurs des registres (pour l'arrêt sur utilisation de variable indicée).

Figure 7 - Forme d'une entrée dans le "DIRECTORY".

c) Structuration du système d'aide à la mise au point.

La construction de ce système a été faite en essayant de diviser le travail en fonctions ; le système d'aide à la mise au point étant lui-même écrit en GSL, chaque fonction se compose d'un ou plusieurs modules GSL.

Ainsi, une fonction particulière exécute chaque commande émise par l'utilisateur ou par le système.

Des fonctions internes, dites de servitude, exécutent les travaux communs à plusieurs fonctions ; ce sont, par exemple, les recherches en table, etc...

d) Dans les systèmes d'aide à la mise au point existants, les requêtes qu'il est possible d'émettre ont une forme définie par une syntaxe. Cette syntaxe ne transparaît pas toujours au niveau de l'implémentation ; assez souvent, la reconnaissance des commandes et leurs traitements se trouvent mélangés.

Dans l'implémentation proposée, le langage de mise au point est un langage régulier [H2]. Il est obtenu par union des automates correspondant aux diverses commandes. Chaque commande est décrite par un langage régulier (type 3). (Voir annexe).

L'adoption de cette méthode permet :

- d'isoler chacune des actions à faire etde faciliter la mise au point globale du système ;
- d'avoir plus de clarté au niveau des actions à effectuer ;
- de pouvoir modifier aisément les commandes par simple changement de la grammaire et des fonctions sémantiques intéressées.

Les commandes restant relativement élémentaires, il apparaît qu'une grammaire de type 3 est suffisante (sauf pour l'affichage et la modification).

Pour chaque commande, nous donnons la grammaire qui lui est associée.

Nous notons :

. Par des phrases entre les signes '<' et '>' les éléments non terminaux.
. Par des grandes lettres entre crochets les mots-clefs (ce sont des éléments terminaux).
. Par des petites lettres entre crochets les éléments terminaux, mais représentant des éléments appartenant au système en construction.
. Le symbole '[⊣]' représente la fin de chaîne.
. Le symbole '|' représente une alternative (il permet de condenser l'écriture de plusieurs règles).

En annexe, nous donnons la grammaire générale et les automates réguliers correspondant à chaque commande.

III-4-2 - IDENTIFICATION

On regroupe sous ce vocable les commandes qui permettent d'accéder au contenu ou à l'adresse des variables du système OBJET.

III-4-2-1 - Localisation de contexte

La localisation de contexte est faite au moyen de la requête "QUALIFY" On peut ainsi accéder à l'ensemble des variables en vie possédées par le système OBJET. Les vérifications effectuées sont relatives à l'activation ou la non-activation du module que l'on désire atteindre.

* La forme syntaxique de la requête est donnée par la grammaire suivante :

Axiome : <qualify fonction>

1 <qualify fonction> → [QUALIFY] <élément qualifié>|[QUALIFY] <fin>

3 <élément qualifié> → [:] <fichier qualifié>|[identificateur de procédure externe] <fin>

5 <fichier qualifié> → [nom de fichier de type debug] <fin>

6 <fin> → [⊣]

L'exécution de cette requête provoque une recherche dans le "DIRECTORY" la recherche s'arrête dès qu'une concordance est rencontrée sur un nom de fichier ou de module.

L'émission de la requête sans paramètre permet de se replacer dans le contexte en vigueur lors de l'arrêt.

III-4-2-2 - Calcul de l'adresse

Cette requête permet d'obtenir l'adresse d'une variable du système OBJET identifiée par un nom symbolique.

Sa forme syntaxique est donnée par la grammaire suivante :

Axiome : <addr-fonction>

1	<addr-fonction>	→	[ADDR] <corps de l'addr>
2	<corps de l'addr>	→	[(]<argument de addr>
3	<argument de addr>	→	[identificateur] <reste de addr>
4	<reste de addr>	→	[(]<indicage d'addr>| [)] <fin>
6	<indicage d'addr>	→	[identificateur] <fermeture d'indice> [constante] <fermeture d'indice>
8	<fermeture d'indice>	→	[)] <fermeture>
9	<fermeture>	→	[)] <fin>
10	<fin>	→	[⊣]

Ceci permet des expressions du type :

ADDR (ID) qui donne l'adresse d'une variable identifiée par 'ID'

ADDR (ID (indice)) qui fournit l'adresse d'un élément de tableau ou d'une sous-chaîne identifiée par 'ID'.

Les formes du type ADDR([identificateur]({[identificateur] → [identificateur]}*)) paraissant peu utilisées n'ont pas été retenues. Le même résultat peut être obtenu en se servant de la commande d'affichage et de la commande 'ADDR'.

III-4-2-3 - Affichage et modification des variables

Ces commandes permettent de connaître ou de modifier le contenu des variables du système OBJET ; elles sont caractérisées par :

* L'accès aux variables se fait de manière symbolique.
* Toute action tient compte du mode attaché à la variable par sa déclaration apparue dans le module "qualifié" (implicitement ou explicitement).
* La forme de ces commandes est un sous-ensemble du langage GSL ; en particulier, elle respecte les conventions adoptées dans le langage GSL pour les conversions d'une valeur d'un mode vers sa représentation dans un autre mode.
* La vérification systématique de la cohérence de la commande vis-à-vis des déclarations des éléments qui y apparaissent, de leur utilisation, de la configuration de la machine sur laquelle s'exécute le système OBJET et du contexte de l'arrêt.

Grammaire de la commande de modification et d'affichage :

Axiome : <affich-modif fonction>

1 <affich-modif fonction> → [identificateur] <affich modif>|[constante] <cst qualification>

3 <affich-modif> → [?] <affich>|[=]<modif>|[(]<indicage>|[→] <qual id>

6 <cst qualification> → [→] <qual id>

7 <qual id> → [identificateur de type basé] <affich modif>

8 <indicage> → [constante] <tabouchaîne>

9 <tabouchaîne> → [:]<offset>|[,] <tableau de chaîne>|[)] <fin de subscript>

12 <offset> → [constante] <fin d'offset expr>

13 <tableau de chaîne> → [constante] <offset expr>

14 <offset expr> → [:] <offset>

15 <fin d'offset expr> → [)] <fin de subscript>

16 <fin de subscript> → [?] <affich>|[=] <modif>|[→] <qual id>

19 <affich> → [⊣]

20 <modif> → [constante] <fin>

21 <fin> → [⊣]

Cette grammaire donne la syntaxe de la commande de modification et d'affichage telle qu'elle fonctionne actuellement.

Elle permet d'afficher ou de modifier des tranches de tableau ou des sous-chaînes par exemple, avec des indices constants.

a) Extensions possibles.

a.1) Nous prévoyons des modifications pour accepter des variables dans les règles 8, 12 et 13.

Ceci donnerait la possibilité d'expression du type

ID1 (ID2 → ID3 (ID4)) ?

qui, actuellement, doit être réalisée en plusieurs commandes.

Nous comptons résoudre ce problème en transformant l'automate régulier correspondant à cette grammaire en automate à pile; nous accepterons donc un langage de type 2. L'état de l'analyse serait sauvegardé dans une pile chaque fois que l'alternative correspondant à un identificateur serait prise dans les règles 8, 12 ou 13. Cette technique d'analyse semble, dans ce cas précis, plus souple que l'utilisation d'un analyseur syntaxique engendré par un transformateur de grammaire [B8] [G2].

Des fonctions sémantiques limiteraient le langage engendré, en particulier afin d'éviter des affectations multiples.

a.2) Nous prévoyons aussi d'accepter, à droite du signe égal, une constante, une variable, une variable qualifiée, la fonction donnant l'adresse d'une variable.

b) Exécution de la commande.

Lorsque l'on veut exécuter la commande, il faut connaître l'adresse de l'élément intéressé, sa longueur et son mode. Ce dernier indique de quelle manière il faut manipuler ce qui se trouve à l'adresse spécifiée.

L'exécution de la commande consiste à faire évoluer un descripteur par l'intermédiaire des fonctions sémantiques contenues dans la grammaire. Ainsi, lorsqu'on parvient à l'une des deux règles 19 ou 21, nous sommes en possession de tous les éléments nécessaires à l'exécution de la commande.

Lorsque le support est le système SPY, le descripteur doit contenir l'adresse translatée. Ceci est effectué par les fonctions sémantiques attachées aux règles 1, 2 et 7. De même, il y a une vérification d'adresse chaque fois que la mémoire de la machine de l'utilisateur est référencée (les translations et les vérifications sont effectuées par une même fonction de servitude).

Exemple : Supposons que l'on ait les déclarations suivantes :

```
DCL        P (10)      POINTER ;
DCL        PP(4)       POINTER BASED ;
DCL        SUBST (20)  CHAR (50)  BASED ;
```

On traite les commandes :

```
P (10)  →  PP (3)  →   SUBST (8,5 : 10) ?
```

et

```
P (10)  →  PP (3)  →   SUBST (8,5 : 10) = 'IL FAIT BEAU'
```

on possèdera le même descripteur jusqu'à l'exécution de l'affichage ou de l'affectation.

L'analyse évolue de la manière suivante :

On spécifie entre parenthèses les règles qui ne font pas progresser le descripteur ; d et d1 sont les multiplicateurs des variables PP et SUBST ; d2 est la taille d'un élément de SUBST.

Règle utilisée	Unité syntaxique	Descripteur		
		adresse	longueur	caractéristique
1 (5)	P	adr(P)	4	PTR.statique.array
8 (11,18)	10	adr(P(10))	4	" " "
7 (5)	PP	Contenu(P(10))	4	" based "
8 (11,18)	3	Contenu(P(10))+3d	4	" " "
7 (5)	SUBST	Contenu(contenu(P(10))+3d)	50	CHAR " "
8 (10)	8	" + 50d1	50	" " "
13 (14)	5	" + 50d1 + 5d2	50	" " "
12 (15)	10	"	10	" " "

La fin de l'analyse se fait en utilisant, dans chacun des cas, les règles 16 et 19, respectivement les règles 17, 20 et 21. Ces règles déterminent le travail à effectuer.

c) Restrictions -

Un certain nombre de restrictions ont été apportées par rapport aux manipulations permises dans le langage GSL.

c.1) Nous ne permettons pas les manipulations de variables de type procédure.

Le langage GSL permet l'utilisation de constantes de type procédure:

Exemple :
```
P : PROC  (R1 REG(1)) ;
  ≀
  ≀
END ;
CALL P ;
```

Il permet aussi l'utilisation de variables de type procédure :

Exemple :
```
DCL   P  PROC  (REG(1))  EXT ;
DCL  (PV1 , PV2)  PROC  (REG(1))  VARIABLE ;
PV1 = P ;
PV2 = PV1 ;
CALL  PV1 ;
```

. Lorsqu'on réalise une affectation de variable de type procédure, on manipule des adresses et l'on vérifie l'identité des modes.

. L'utilisateur veut connaître le contenu d'une variable de type procédure ; il émet la commande PV1 ?
Il est nécessaire de rechercher dans le "DIRECTORY" le nom de la procédure dont l'adresse est contenue dans PV1, soit P.

Si la commande : PV1 = P est émise, il faut rechercher dans le DIRECTO l'adresse de P et, ensuite, comparer les modes de PV1 et de P avant d'effectuer l'affectation. Ce processus (réalisé dans le chargeur assisté) est relativement coûteux ; des manipulations de ce type ne sont pas permises.

c.2) Nous ne permettons pas d'affectations multiples.

Le langage GSL permet de réaliser des affectations multiples :

Exemple : Soit les déclarations :

```
DCL   CH1      CHAR (10) ;
DCL   CH2 (8)  CHAR (20) ;
DCL   BY1 (4)  BYTE (8)  ;
```

En se servant de "l'affectation à droite" (⇒), on peut écrire :

'A' ⇒ BY1 (4,4 : 1) ⇒ CH2 (5,2 : 1) ⇒ CH1 (3 : 1)

Supposons qu'une telle forme soit permise. L'évaluation se fait de gauche à droite ; l'exécution de la commande a lieu au fur et à mesure de la progression de l'analyse.

Si une erreur est détectée au cours de l'évaluation, il peut s'ensuivre de désastreux effets de bord, dans le cas où des affectations ont déjà été réalisées.

Si l'on veut se prémunir contre de tels accidents, il faut instaurer une sauvegarde des éléments touchés et restaurer leur état initial en cas

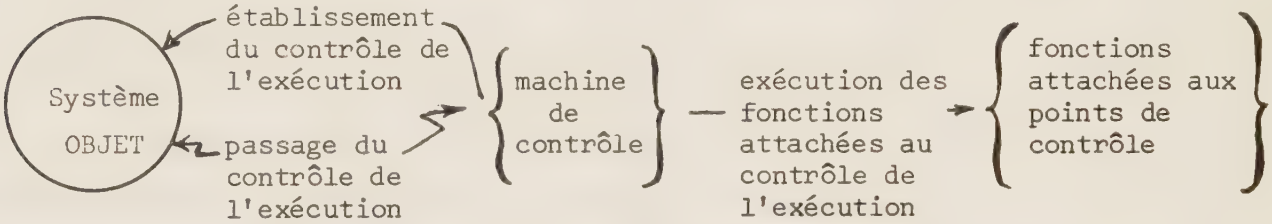
d'erreur, ou faire plusieurs passages et n'exécuter la commande que lorsqu'elle est correcte.

Les deux méthodes sont lourdes et coûteuses. Nous ne permettons pas ce type d'affectation.

III-4-3 - Machine de contrôle

On appelle ainsi l'ensemble des mécanismes permettant de contrôler l'exécution du système OBJET.

L'aspect important vient du fait qu'il n'y a qu'un seul mécanisme de contrôle de l'exécution. Ensuite, sont attachées au point de contrôle des fonctions plus ou moins complexes, disjointes du mécanisme de contrôle, émises par l'utilisateur ou par le système de mise au point assistée pour suivre le déroulement de l'exécution.



Il y a dissociation entre la machine de contrôle et les fonctions attachées au contrôle. Ces dernières peuvent varier, leur nombre augmenter ou diminuer, peu importe, la structure du système reste la même.

III-4-3-1 - Le contrôle d'exécution

Il s'agit, par un moyen quelconque, de détourner l'exécution du système OBJET au profit du système de mise au point. Ce contrôle qu'exerce le système de mise au point est transparent pour le programmeur.

La manière d'exercer le contrôle dépend du calculateur et des buts que l'on assigne à l'outil d'aide à la mise au point.

Dans notre cas, le choix s'est porté sur la création d'une interruption programme qui, par la suite, est interprétée.

Le processus est le suivant :

* On détermine dans le code objet l'instruction machine où doit s'exercer le contrôle.
* On sauvegarde le code opération de l'instruction machine.
* On remplace le code de l'instruction machine par un code invalide.

Lorsqu'une interruption programme survient :

* On détermine le point où elle est survenue.
* Si elle correspond à un contrôle précédemment mis, elle est prise en compte.
* Si non, elle est reflétée vers l'utilisateur.

La permanence du contrôle est assurée par la simulation de l'instruction machine sur laquelle s'exerce le contrôle.

Une meilleure compréhension du mécanisme est possible lorsqu'on connaît les tables nécessaires.

III-4-3-2 - Table de contrôle et de fonctions attachées au contrôle

A chaque module ayant une entrée dans le "DIRECTORY" est associée une table de contrôle et de fonctions. Deux objectifs lui sont assignés :

. Permettre le fonctionnement du contrôle.

. Attacher des fonctions aux différents points du système OBJET où s'exerce le contrôle.

Cette table possède autant d'entrées qu'il y a d'instructions exécutables dans le module en cause. La forme d'une entrée, dans la table de contrôle et de fonctions,est donnée par la figure 9. Chaque entrée comporte :

- Une partie contrôle se résumant au champ (1) et contenant la valeur du code opération substitué si un contrôle est voulu, zéro dans le cas contraire.
- Les champs (2) (3) (4) (5) et (6) étant des renseignements sur les fonctions à accomplir.

L'utilisation de cette table se fait conjointement avec la table de correspondance décrite dans le paragraphe III - 2. Nous en rappelons la forme dans la figure 8.

Le mécanisme de la mise en place d'un contrôle est évident, celui de sa reconnaissance est décrit par la figure 10.

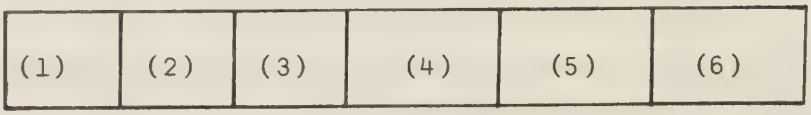
Figure 8 - Table de correspondance.

(1) numéro de l'instruction GSL source.

(2) déplacement de la 1ère instruction assembleur générée.

(3) déplacement du descripteur des registres de l'optimisation.

Figure 9 - Table de contrôle et de fonctions.



(1) Code opération substitué.

(2) Déplacement d'un compteur dans le cas d'un arrêt sur nombre de passages.

(3) Compteur du nombre d'arrêts sur référence identificateur portés par l'instruction.

(4) Déplacement d'un descripteur d'étiquette si l'arrêt a été demandé en spécifiant une étiquette.

(5) Description des fonctions à accomplir (dans l'ordre de priorité).

- arrêt de mise à jour de la pile en entrée
- " " " " " en sortie
- macro-instruction
- test de compteur et incrémentation
- arrêt utilisateur inconditionnel.

(6) Index de la macro s'il en existe une.

Figure 10 - Etapes principales de la reconnaissance d'un point de contrôle.

. Connaissant l'adresse absolue de l'interruption de type programme, on localise dans le "DIRECTORY" le module où elle s'est produite. On calcule alors l'adresse de l'instruction qui a provoqué l'interruption par rapport au début du module.

. Si l'interruption n'appartient à aucun des modules décrits dans le "DIRECTORY" (modules en assembleur par exemple), elle est reflétée vers l'utilisateur.

. On recherche ensuite dans la table de correspondance s'il s'agit d'un début d'instruction GSL (égalité du champ (2) de la table et de l'adresse relative de l'instruction interrompue). Si la recherche échoue, on calcule le numéro de l'instruction GSL où s'est produite l'interruption, puis on reflète le numéro de l'instruction GSL et l'interruption vers l'utilisateur.

. La recherche ayant abouti, on teste si un contrôle avait été placé sur cette instruction (à l'aide de la table de contrôle et de fonctions). Si ce n'est pas le cas, le numéro de l'instruction GSL et l'interruption sont reflétés vers l'utilisateur.

. Si un contrôle avait été placé en ce point, on exécute l'ensemble des fonctions attachées à ce point de contrôle.

III-4-3-3 - Fonctions attachées aux points de contrôle

Nous allons décrire les différentes fonctions que l'on peut attacher à un point de contrôle. La liste fournie n'est pas limitative ; un utilisateur plein d'imagination pourrait en concevoir d'autres, leur ajout n'influençant en rien le contrôle.

On sépare les fonctions en deux classes : celles accessibles à l'utilisateur et celles du système d'aide à la mise au point.

III-4-3-3-1 - Fonction du système d'aide à la mise au point

a) La pile de contrôle.

Elle permet de connaître, à un instant donné, l'ensemble des modules actifs et de décider ainsi de la validité de l'accès aux variables d'un module.

Chaque entrée correspond à un module et contient :

* Le déplacement dans le "DIRECTORY" de l'entrée relative au module.

* Le déplacement dans la table des identificateurs du descripteur du module.

b) Les fonctions.

Nous avons deux fonctions attachées respectivement à l'entrée et à la sortie de chaque module. Leur but est de tenir à jour la pile d'exécution et, accessoirement, de permettre à l'utilisateur, s'il le désire, d'arrêter l'exécution à l'entrée et à la sortie d'un module.

III-4-3-3-2 - Fonctions accessibles à l'utilisateur

Une instruction peut supporter plusieurs fonctions. Le traitement de ces fonctions se fait suivant un ordre prédéfini, à savoir : exécution de macro requête, arrêt sur erreur d'indiçage, arrêt conditionnel, arrêt sur référence, arrêt inconditionnel.

Précisons les différentes fonctions que l'utilisateur peut rattacher à une instruction.

a) Exécution d'une macro instruction.

Il s'agit là de faire exécuter automatiquement une séquence de commandes de mise au point, lors de chaque passage en cette instruction.

L'accès à la séquence de commande se fait par l'intermédiaire d'une table.

b) Arrêt conditionnel.

L'arrêt est conditionné par le nombre de passages de l'exécution au point spécifié. Ceci est réalisé en attachant un compteur à l'instruction en cause, en l'incrémentant et en le testant à chaque passage. Les compteurs sont initialement à zéro, leur nombre est limité, la valeur à tester et le test sont rattachés au compteur. Il y a une table de compteur par module.

c) Arrêt inconditionnel.

Le contrôle est rendu à l'utilisateur ; le système de mise au point attend des directives.

d) Nous avons prévu d'implémenter deux autres types d'arrêts.

* Arrêt sur référence.

L'arrêt a lieu lorsqu'on fait référence à un nom de variable. (Il s'agit des règles 2, 5, 25 et 26; cf. figure 11).

A l'aide de la table des références croisées, on peut connaître les instructions où est référencé un identificateur dans un module. Une table, possédant autant d'entrées qu'il existe d'identificateurs référencés dans le module, nous permet de connaître les identificateurs sur lesquels porte l'arrêt.

Le champ (3) d'une entrée de la table de contrôle et de fonctions compte le nombre d'arrêts sur référence portés par l'instruction (limité à 7).

* Arrêt sur erreur d'indiçage.

L'arrêt a lieu lorsque la valeur d'un indice n'est plus contenue entre les bornes d'un tableau ou définit une sous-chaîne n'appartenant plus à une variable. A chaque utilisation d'un tel élément est attaché un descripteur. Une table semblable à la table de contrôle et de fonctions nous permet d'arrêter l'exécution et de vérifier, sur demande de l'utilisateur, la validité de l'indiçage. Le descripteur contient le déplacement de l'instruction machine où doit être fait le test, la valeur de l'indice ou un moyen de le retrouver, et l'élément indicé.

III-4-3-4 - Forme externe des requêtes de contrôle accessibles à l'utilisateu

Il s'agit de la manière dont l'utilisateur va exprimer les fonctions décrites précédemment. On distingue deux types de requêtes du contrôle de l'exécution accessibles à l'utilisateur :

- Les requêtes de contrôle ponctuel.
- Les requêtes de trace.

III-4-3-4-1 - Les requêtes de contrôle ponctuel

La forme syntaxique de ces requêtes est donnée par la grammaire de la figure 11. Nous qualifions le contrôle de "ponctuel" car il se rapporte à une instruction ou un groupe d'instructions. L'instruction peut être désignée par son numéro, ses étiquettes, ou par une propriété (faire référence à un identificateur, sortie de procédure, etc...). Lorsqu'on se sert d'une propriété

dans l'expression d'une requête, on désigne un groupe d'instructions (sauf lorsque la requête est relative à l'activation ou la désactivation d'une procédure). La fonction est un arrêt (STOP) pondéré par une condition (IF) ou effectué après l'exécution d'une macro-instruction ; l'utilisateur peut désirer ne pas reprendre le contrôle (RUN).

L'exécution est possible en pas à pas (ALL) ou arrêtée lors d'une référence à un identificateur (REFERENCE), ou lors d'une activation ou désactivation d'une procédure externe (ENTPROC, EXITPROC).

* Il est à remarquer qu'un "macro langage" plus élaboré qu'un simple regroupement de commandes de mise au point, (c'est-à-dire un macro langage permettant de faire des tests, possédant des instructions propres) éviterait la règle 10 de la grammaire, en traitant les manipulations de compteurs.

* Les fonctions peuvent être cumulées, leur exécution étant ordonnée. La pondération par un IF porte sur le retour du contrôle vers l'utilisateur et non sur l'exécution d'une macro-instruction s'il en existe une. Pour que le contrôle soit rendu au programmeur, il suffit qu'une fonction demande un "STOP".

III-4-3-4-2 - Trace

Il existe deux types de trace ; tous deux se servent du contrôle interne effectué, sur l'exécution, par le système de mise au point assistée.

a) La commande "SUBTRACE" permet de connaître l'ensemble des modules actifs à un instant donné ; les renseignements sont contenus dans la pile d'exécution.

b) La commande "TRACE" permet d'avoir la liste des modules activés et désactivés au cours de l'exécution. Ces renseignements sont disponibles lors de chaque mise à jour de la pile d'exécution.

Figure 11 - Grammaire de la requête d'arrêt.

Axiome : <control fonction>

1	<control fonction>	→	[AT] <corps d'arrêt>|[TRACE] <fin>|[SUBTRACE]<fin
4	<corps d'arrêt>	→	[:] <spécial arrêt>|[numéro d'instruction] <action sur arrêt>|[identificateur de type étiquette] <action sur arrêt>
7	<action sur arrêt>	→	[STOP] <cond mac>|[RUN] <mac seule>
9	<cond mac>	→	[MACRO] <mac name>|[IF] <cond expr>| <—1>
12	<mac name>	→	[nom de macro] <fin>
13	<cond expr>	→	[COUNT] <rest expr>
15	<rest expr>	→	[>]<val comp>|[=]<val comp>|[<]<val comp>
17	<val comp>	→	[constante entière] <fin>
18	<mac seule>	→	[MACRO] <mac name>
19	<spécial arrêt>	→	[ENT PROC] <exproc name>|[EXITPROC] <exproc name> [REFERENCE] <iderefe>|[SUBSCRIPT] <subscriptuse>|[ALL] <borne all>
24	<exproc name>	→	[identificateur de procédure externe] <exproc name>|[identificateur de procédure externe] <ent sort action>
26	<ent sort action>	→	[STOP] <fin>
27	<idrefe>	→	[identificateur autre qu'étiquette et procédure] <refe action>
28	<refe action>	→	[STOP] <fin>
29	<subscriptuse>	→	[identificateur de variable indicé] <fin>
30	<borne all>	→	[FROM] <borne un>
31	<borne un>	→	[numéro d'instruction] <borne deux>| [identificateur de type étiquette] <borne deux>
32	<borne deux>	→	[TO] <val borne deux>

34 <val borne deux> → [numéro d'instruction] <all action>
[identificateur de type étiquette] <all action>

36 <all action> → [RUN] <mac seule> | [STOP] <mac all>

38 <mac all> → [MACRO] <mac name> <⊣>

40 <fin> → <⊣>

III-4-4 - Commandes diverses

Ce sont des commandes qui permettent d'annuler l'effet de commandes précédemment émises ou d'enquêter sur l'état du système OBJET.

III-4-4-1 - Annulation et abandon

a) La commande "QUIT" permet d'abandonner la mise au point du système OBJET.

b) La commande "RESET" permet d'annuler l'effet de certaines fonctions. Sa forme syntaxique est donnée par la figure 12.

* Il est possible d'enlever un compteur, une macro commande ou un stop attaché à une instruction, sans altérer les autres fonctions ; le contrôle de l'instruction est abandonné lorsqu'aucune fonction n'y est plus attachée.

* De même, il est possible d'agir globalement mais toujours au niveau du module qualifié. RESET ALL toutes les fonctions intéressant le module sont enlevées (macro, etc...).

Cependant il est possible de faire une sélection sur les macros d'un nom donné, ou les arrêts sur compteur, ou les arrêts inconditionnels.

* Il est possible de supprimer le pas à pas :

RESET AT ALL STOP

(ce qui ne supprime pas l'exécution d'une macro commande s'il y en a une

RESET AT ALL MACRO

permet de la supprimer

RESET AT ALL

supprime le mode pas à pas et ce qui y est rattaché.

Figure 12 - Grammaire de la requête d'annulation.

Axiome : <Reset fonction>

1	<Reset fonction>	→	[RESET] <reset argument>
2	<reset argument>	→	[AT] <at all argument>|[ALL] <fonction>| [TRACE] <fin>|[MACRO] <localisation de at>| [COUNT] <localisation de at>|[STOP]<porteur d'arrêt> [SUBSCRIPT] <subscript use>
9	<at all argument>	→	[ALL] <fin>|[ALL] <action du all>
11	<action du all>	→	[STOP] <fin>|[MACRO] <fin>
13	<fonction>	→	[MACRO] <fin>|[MACRO] <mac name> [STOP] <fin>|[COUNT] <fin>
17	<mac name>	→	[nom de macrocommande] <fin>
18	<localisation de at>	→	[AT] <localisation expression>
19	<localisation expression>	→	[identificateur de type étiquette] <fin> | [numéro d'instruction] <fin>
21	<porteur d'arrêt>	→	[AT] <localisation expression>|REFERENCE| <reset réfé>| [ENTPROC] <liste proc>|EXIT PROC] <listeproc
25	<liste proc>	→	[nom de procédure externe] <liste proc>| [nom de procédure externe] <fin>
27	<reset réf>	→	[identificateur autre qu'étiquette et procédure] <fin>
28	<fin>	→	<⊣>

III-4-4-2 - Enquête - Groupe de commandes - Reprise de l'exécution

a) La commande "QUERY" permet de connaître les fonctions attachées par l'utilisateur aux instructions d'un module (le module qualifié). Sa forme syntaxique est donnée par la grammaire de la figure 13.

On peut ainsi demander l'ensemble des instructions ayant une fonction d'arrêt inconditionnel, un compteur, une macro ou une macro de nom donné.

Au niveau du système OBJET, l'enquête porte sur les arrêts d'activation, désactivation de module, ou de pas à pas.

Figure 13 - Grammaire de la commande d'enquête.

Axiome : <query fonction>

1	<query fonction>	→	[QUERY] <corps du query>
2	<corps du query>	→	[COUNT] <count query>|[MACRO] <mac name>| [STOP] <fin>| [ENTPROC] <fin>|[EXITPROC] <fin>|[REFERENCE] <fin>| [AT] <at all query>|[MACRO] <fin>
10	<count query>	→	[AT] <localisation expression>|[ONLY] <fin>
12	<localisation expression>	→	[identificateur de type étiquette] <fin>| [numéro d'instruction] <fin>
14	<mac name>	→	[nom de macro commande] <fin>
15	<at all query>	→	[ALL] <fin>
16	<fin>	→	<⊣>

b) La création d'une macro commande se fait pár :

* L'intermédiaire de la commande AT
* L'ordre 'MACRO" suivi d'un nom de macro commande ; le système lit les lignes jusqu'à rencontrer ENDMAC qui spécifie la fin de la macro commande.
* Lecture de la macro commande à partir du disque par l'instruction GET suivie d'un nom de macro commande. Il y a recherche d'un fichier portant ce nom.

La suppression d'une macro commande se fait par l'intermédiaire de l'ordre "KILL" suivi d'un nom de macro.

Les noms des macro commandes sont placés dans une table, un index correspond à chacune d'elles. Le nombre de macro commandes utilisables pendant une session est limité (taille maximum de l'index).

Afin d'éviter des recherches coûteuses, le fait de tuer une macro ne libère pas l'entrée, dans la table, mais l'invalide ; la remise à jour des arrêts portant une fonction macro est faite lors d'une tentative d'exécution d'une macro commande invalidée.

c) L'utilisateur a la possibilité de reprendre le contrôle au moyen de la touche "attention" disponible sur les télétypes.

d) La reprise de l'exécution se fait au moyen de la requête "GO". Les problèmes soulevés par la reprise de l'exécution par un "GOTO" seront abordés au cours du chapitre suivant.

e) Nous pensons donner la possibilité d'activer une procédure GSL (commande CALL). On vérifierait alors la cohérence de l'appel.

III-4-5 - Technique en vue d'une implémentation efficace

Les modifications que nous décrivons ont eu pour seul but de tenir compte du système support de l'outil proposé (CP67 [I8]) et des méthodes de travail des utilisateurs.

On peut, en effet, remarquer que :

- le système CP67 simule des machines réelles et utilise le mécanisme de pagination du calculateur S 360/67. Il semble évident que nous augmentons le temps de réponse en accroissant le nombre de pages référencées (mécanisme de "faute de page").
- l'utilisateur se sert, en général, d'un petit nombre de points de contrôle lorsqu'il cherche à mettre un système au point.
- un programme source comporte plus d'instructions exécutables (qui donnent naissance à du code objet exécutable) que d'instructions non exécutables (le rapport est d'environ 1/3).

De ces constatations, il ressort que :

- La manipulation de tables engendre un grand nombre de "fautes de pages", d'où la certitude d'un mauvais temps de réponse.
- Le coefficient de remplissage des tables est faible.
- La reconnaissance d'un point de contrôle est relativement longue.

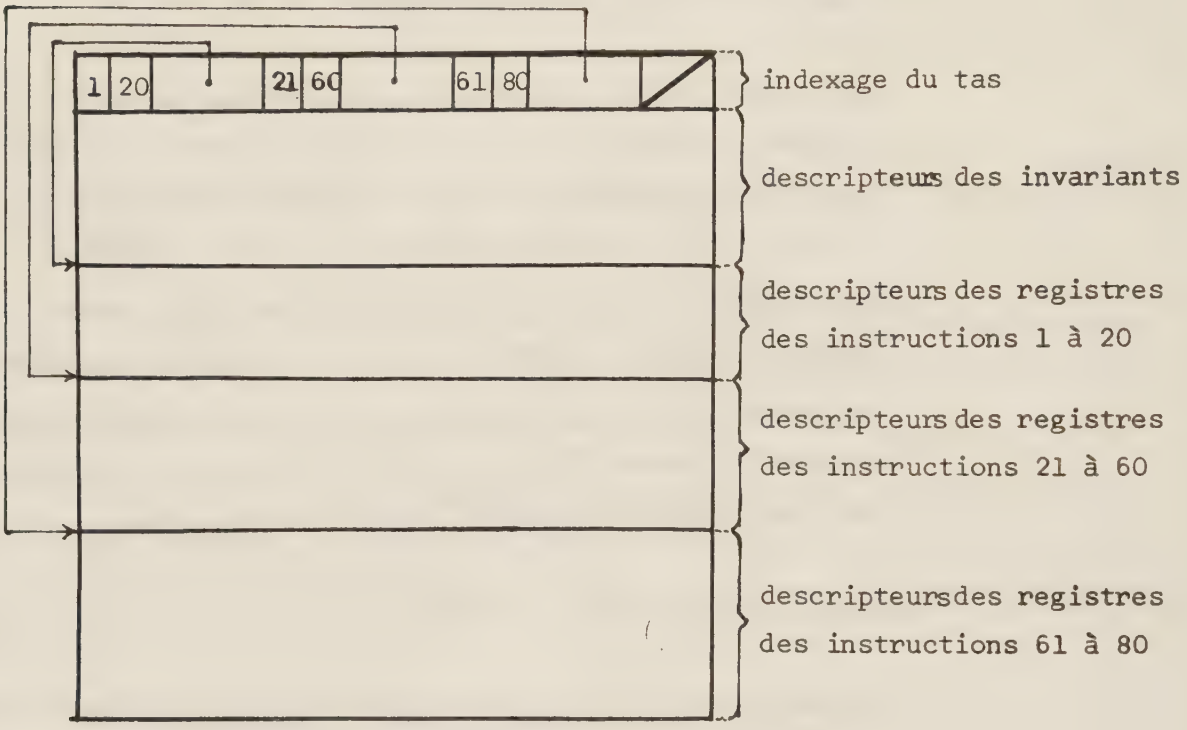
Ceci nous a incité à modifier la structure des données fournies par le compilateur,de la manière suivante :

a) La table de correspondance ne contient plus que les déplacements des instructions machine engendrées. L'accès se fait directement en indexant la table par le numéro de l'instruction GSL.

b) L'ensemble des descripteurs des registres est légèrement modifié. On tient compte du fait qu'en certains points d'un programme GSL, l'ensemble des regis-

tres gérés par le compilateur est vide (étiquette, appels de certaines procédures).

Exemple - Dans un programme GSL, l'ensemble des registres gérés par le compilateur est vide au niveau des instructions 1, 20, 60 et 80. L'ensemble des descripteurs des registres a, sur disque, la structure suivante :



L'accès au descripteur des registres attaché à l'instruction i se fait de la manière suivante :

. On recherche le groupe correspondant à l'instruction i

. On constitue un descripteur des 16 registres

. On fait évoluer ce descripteur jusqu'au moment où l'on atteint l'instruction i.

Cette organisation supprime le champ (3) (cf. III.2) du descripteur des registres attaché à une instruction, puisque l'accès est maintenant séquentiel.

Le tas des registres n'est plus référencé à partir de la table de correspondance, donc manipulé de manière indépendante.

c) La table de contrôle et de fonctions.

Nous tenons compte, ici, du mécanisme de pagination et du petit nombre de points où doit s'exercer le contrôle. Nous structurons la table de contrôle et de fonctions, pour chaque module, en une liste de descripteurs de contrôle et de fonctions répartie en pages.

Tout adressage se fait sur deux octets, soit 16 bits.

. 12 bits permettent de spécifier un déplacement dans une page.

. 4 bits permettent de spécifier le numéro de la page.

Lorsqu'un contrôle est demandé, un élément (cf. figure 14) est ajouté à la liste des descripteurs de contrôle et de fonctions chaînés à partir du "DIRECTORY". Un module spécialisé du système d'aide à la mise au point accomplit les fonctions de :

. Recherche d'un élément de la liste sur critère de numéro d'instruction GSL ou déplacement d'instruction assembleur.

. Gestion des pages contenant ces listes.

En retour, ce module fournit l'adresse réelle du descripteur de contrôle et de fonctions.

A chaque descripteur de contrôle et de fonctions, on lui associe, si cela est nécessaire, le descripteur des registres attaché à l'instruction (cf. figure 15). Les descripteurs des registres sont reconstitués comme il a été précédemment décrit ; leur gestion est assurée par un mécanisme de pages semblable à celui que l'on vient de voir (cf. figure 16). Les descripteurs des registres sont de taille fixe, facilitant ainsi leur gestion. Lorsque le descripteur d'un point de contrôle et de fonctions est libéré, le descripteur des registres associé à ce point de contrôle est aussi libéré.

d) Avantages et inconvénients.

Les avantages principaux sont les suivants :

. Une page contient 292 descripteurs de contrôle et de fonctions ; donc on est quasiment sûr d'avoir un regroupement dans une même page de tous les descripteurs de contrôle et de fonctions intéressant un même module.

. Il n'y a pas de place perdue puisque, à chaque élément de la liste, correspond un descripteur actif de contrôle et de fonctions.

Les inconvénients sont les suivants :

. Cette technique perd de son efficacité si le nombre de points de contrôle est trop important.

. Si l'utilisateur veut une bonne efficacité, il doit regrouper, dans le temps, l'acquisition des descripteurs de contrôle et de fonctions.

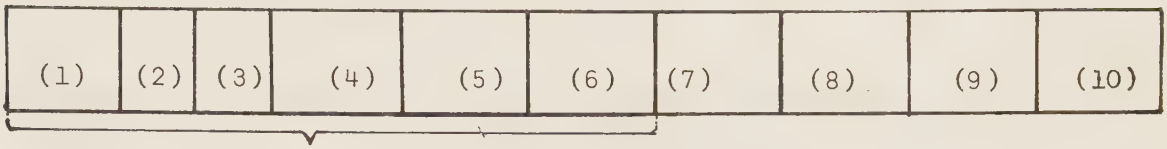
. Le pas à pas est un mode très inefficace, aussi avons-nous modifié la syntaxe de la commande "AT" afin d'avoir un pas à pas dans une portion de module.

Exemple : On veut faire un pas à pas entre les instructions n1 et n2 d'un module ; on émet la commande:

AT : ALL FROM n1 TO n2 STOP

. L'utilisateur, positionnant et dépositionnant des arrêts trop fréquemment, se trouve pénalisé par la méthode adoptée.

Figure 14 - Forme d'un descripteur de contrôle et de fonctions.



partie inchangée par rapport à la structure en table. Voir figure 1 paragraphe III.4.3.2.

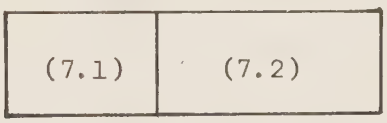
(7) lien vers l'élément suivant dans la liste.

(8) numéro de l'instruction GSL sur laquelle porte le contrôle.

(9) déplacement vers le descripteur des registres attaché à l'instruction.

(10) déplacement de l'instruction assembleur sur laquelle s'exerce le contrôle. Ce déplacement est obtenu à partir de la table de correspondance, lors de la mise en place du point de contrôle.

* Le champ (7) a la structure suivante :

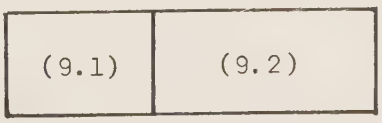


(7.1) Constitué de 4 bits qui donnent le numéro de la page. Le nil est égal à la valeur 0.

(7.2) Constitué de 12 bits qui fournissent l'adresse dans la page.

Cette gestion permet d'avoir 16 pages de 4K ; ceci correspond à 4672 points de contrôle au maximum, et au même instant.

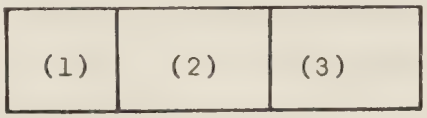
* Le champ (9) a la structure suivante :



Les champs (9.1) et (9.2) ont la même signification que précédemment ; le nil est l'octet 0 de la page. On a, ainsi, 16 pages de 4096 octets.

Figure 15 - Forme d'un descripteur des registres attaché à un descripteur de contrôle et de fonctions.

Chaque descripteur contient 16 éléments (un par registre), semblables à celui dont la description suit :



(1) Donne la signification des champs (2) et(3) ; la signification est la suivante :

. Registre système ou global (c'est-à-dire invariant)

. Registre géré par l'utilisateur

. " " " le compilateur et significatif.

Dans ce dernier cas, on a les possibilités suivantes :

- Contient la valeur d'une variable simple
- " " " variable indicée
- " une expression du type I × Cst
- " " " " (I - 1) × Cst.

(2) et (3) contiennent des pointeurs vers les descripteurs des variables impliquées dans l'expression ou des valeurs de constantes, suivant le contenu du champ (1).

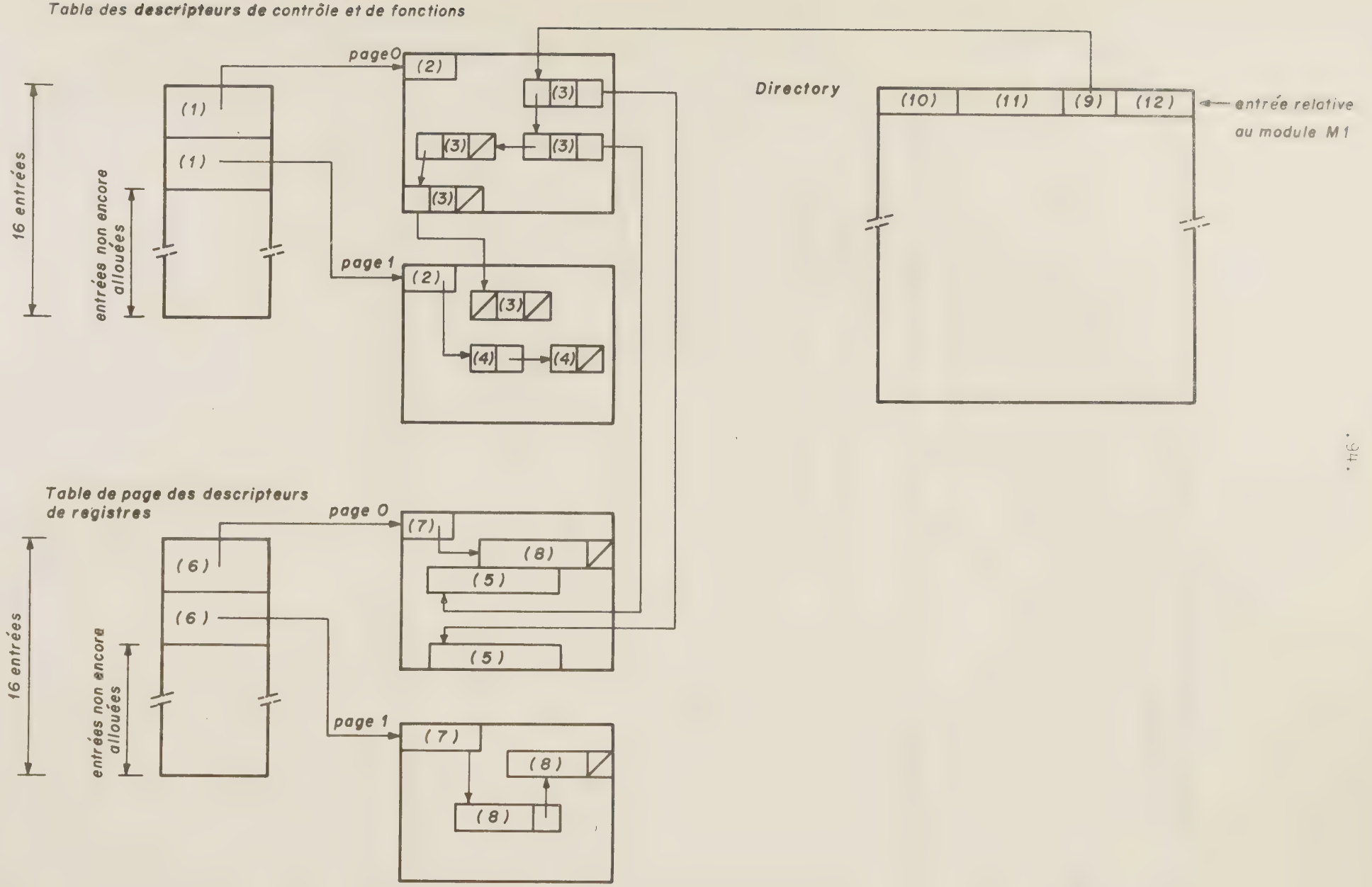


Table des descripteurs de contrôle et de fonctions
page0
(2)
(3)
(1)
(1)
(3)
(3)
(3)
16 entrées
entrées non encore allouées
page 1
(2)
(3)
(4)
(4)
Directory
(10)
(11)
(9)
(12)
entrée relative au module M1
Table de page des descripteurs de registres
page 0
(7)
(8)
(5)
(5)
(6)
(6)
16 entrées
entrées non encore allouées
page 1
(7)
(8)
(8)

Figure 16 : Organisation des descripteurs de contrôle et de fonctions et des descripteurs des registres associés.

Le module M1 comporte 5 points de contrôle et de fonctions dont deux ont nécessité la connaissance des registres associés.

Description de la figure 16 :

(1) Adresse réelle de la page des descripteurs de contrôle et de fonctions.

(2) Tête de la liste des éléments libres dans la page des descripteurs de contrôle et de fonctions. Elle contient :

. Le déplacement du premier élément libre dans la page.
. Le nombre d'éléments libres dans la page.

(3) Descripteur de contrôle et de fonctions (le premier élément de la liste est adressé à partir du "DIRECTORY" ; la forme d'un élément est fourni par la figure 14).

(4) Elément libre dans la page des descripteurs de contrôle et de fonctions (ces éléments sont chaînés entre eux à l'intérieur d'une même page).

(5) Descripteur des registres attaché à l'instruction sur laquelle porte le contrôle (il est présent seulement si sa consultation s'est avérée nécessaire).

(6) Adresse réelle de la page des descripteurs de registres.

(7) Tête de la liste des éléments libres dans la page des descripteurs de registres. Elle contient :

. Le déplacement du premier élément libre dans la page.
. Le nombre d'éléments libres dans la page.

(8) Elément libre dans la page des descripteurs de registres.

(9) Tête de la liste des descripteurs de contrôle et de fonctions attachés au module (il se compose d'un numéro de page et d'un déplacement dans la page, soit 16 bits).

(10) Nom du fichier contenant la description du module M1.

(11) Description de l'implantation en mémoire du module M1.

(12) Adresse des tables et noms attachés au module M1.

Les champs (10) (11) et (12) ont été décrits en détail au cours du paragraphe III-4-1 (figure 7).

CHAPITRE IV

CRITIQUE DU SYSTEME PROPOSE ET EVOLUTION FUTURE

Au cours de ce chapitre, nous nous proposons de comparer la méthode adoptée avec d'autres méthodes existantes, de voir les supports qui lui sont nécessaires, de décrire les problèmes majeurs, enfin d'envisager des extensions futures.

IV-1 - COMPARAISON AVEC DES SYSTEMES EXISTANTS

Comme on a pu s'en rendre compte, la construction d'un outil d'aide à la mise au point présente deux problèmes fondamentaux :

. Le problème de l'identification qui permet de rattacher à un identificateur une variable avec son mode.

. Le mécanisme du contrôle de l'exécution.

On n'envisage pas le problème inverse de l'identification qui, à une adresse, fait correspondre l'ensemble des identificateurs qui lui sont rattachés (Ce problème n'est résolu que dans le cas d'un arrêt où l'on doit identifier l'instruction source qui porte l'arrêt).

Comparons donc la méthode adoptée pour résoudre ces problèmes et celles prises dans des systèmes tels que FORTBUG [J2] et dans un compilateur PL/1 ou ALGOL 60 incrémental [B4] [G3].

IV-1-1 - Caractéristiques principales des deux systèmes

Rappelons sommairement les principaux aspects de ces deux systèmes.

a) Dans un compilateur incrémental, le programme utilisateur est analysé syntaxiquement incrément par incrément (l'incrément est l'unité d'édition, de compilation et d'exécution) ; chaque incrément donne naissance à :

- une entrée dans une table ou dictionnaire de contrôle.
- un pseudo-code traduction compacte de la signification de l'incrément et des objets qui y sont manipulés.
- chaque identificateur apparu est répertorié dans une table (une entrée par nom distinct).

L'exécution se fait en interprétant le pseudo-code. Il y a possibilité de découpage d'un programme en "Segments", les liens étant vérifiés à l'exécution.

L'indépendance des incréments entre eux constitue un point très important.

b) Dans le système FORTBUG (destiné à la mise au point de programmes écrits en FORTRAN) le code objet est du code machine exécutable. Sur option seulement, le compilateur procède de la manière suivante :

- Au début de chaque groupe de code objet correspondant à une instruction source, il génère une séquence de transfert vers un superviseur d'exécution et des renseignements (n° d'étiquette de l'instruction, activation ou désactivation de la routine).
- Il force la génération de la table des identificateurs avec leur mode (par l'intermédiaire d'une "name list").

IV-1-2 - Problème de l'identification

a) Rappel du problème

Un identificateur repère un ensemble de variables ; on veut, à partir d'un identificateur :

- Pouvoir désigner l'une des variables de l'ensemble des variables repérées.
- Connaître ses attributs.
- Accéder à sa valeur.

b) Conservation des caractéristiques

- Dans FORTBUG ainsi que pour l'outil proposé, ces caractéristiques sont conservées dans la table des identificateurs associée au module.
- Dans le compilateur incrémental, elles sont dans le pseudo-code et servent à l'interpréteur pour l'allocation de leur emplacement mémoire.

c) L'accès et sa validité

- Dans FORTBUG ainsi que dans le système proposé, une pile de contrôle est mise à jour par le système d'aide à la mise au point à chaque activation et désactivation de routine. Pour atteindre une variable, il suffit de préciser son nom et le nom du module. La pile de contrôle (différente de la pile d'exécution dans un langage possédant la notion de variables dynamiques) indique si le module est actif et si la variable a un sens ; la table des identificateurs donne son adresse et ses attributs.

- Dans un compilateur incrémental, l'interpréteur met à jour, lors de chaque activation ou désactivation de blocs, la pile d'exécution (elle contient les variables) ainsi que la table des symboles.

Un nom d'identificateur fournit une entrée dans la table des symboles à partir de là, on trouve sur la pile d'exécution, chaînées, les différentes variables en vie avec leurs attributs, la plus récente étant la première de la chaîne. Dans ce cas, pour atteindre une variable, il faut préciser son nom, le numéro de bloc et le niveau de récursivité.

On se rend compte qu'à la récursivité près, les principes sont similaires ; on précise un contexte qui permet d'avoir une relation univoque entre le nom et la variable. Le contexte dépend du langage : ainsi, en FORTRAN, le nom du module suffit à le préciser ; en ALGOL 60, il faut, en plus, le numéro de bloc et le niveau de récursivité.

IV-1-3 - Contrôle de l'exécution

Il permet au système de mise au point assistée de reprendre le contrôle La mise en place d'un point de contrôle comporte deux temps :

. Identifier l'instruction que l'on veut contrôler.
. Se souvenir qu'il faut détourner l'exécution du système en construction en ce point.

L'identification d'un point de contrôle implique la reconnaissance de l'instruction source intéressée (identification inverse).

a) Mécanisme de FORTBUG

Une table de contrôle permet de se souvenir des points où l'exécution doit être arrêtée.

Un superviseur d'exécution reprend le contrôle avant l'exécution de chaque instruction du programme FORTRAN. L'identification inverse ainsi que les fonctions de mise à jour de la pile de contrôle et d'arrêt sont effectuées grâce à l'information contenue dans le code objet.

b) Mécanisme d'un compilateur incrémental

Le dictionnaire de contrôle indique à l'interpréteur, pour chaque instruction, l'adresse du pseudo-code, le type de l'instruction et l'index dans la même table de la prochaine instruction à exécuter ; on peut rajouter un indicateur permettant de se souvenir qu'il y a un point de contrôle.

c) Mécanisme du système proposé

On possède un superviseur d'exécution, mais la reprise du contrôle n'est effectuée que lorsqu'elle a été demandée et non systématiquement.

Les descriptions des instructions ne sont pas incluses dans le code ; il s'en suit qu'un calcul est nécessaire pour retrouver la description. On procède à une identification inverse qui, à partir d'une adresse, nous fait retrouver le ou les identificateurs de l'instruction (numéro et étiquettes).

Pour se souvenir du point de contrôle, on procède à la mise en place d'un transfert vers le superviseur d'exécution et à l'enregistrement du point de contrôle (Création d'une interruption programme, positionnement dans la table).

Ces trois solutions sont différentes ; l'intérêt des deux premières est la simplicité de la reconnaissance d'un arrêt.

La méthode adoptée évite cependant de pénaliser, par des retours vers un superviseur d'exécution ou par une interprétation, des programmes ne possédant pas de point de contrôle ; de plus, elle ne fait pas varier la place mémoire occupée par le système objet. Cependant, si un programme contient un nombre élevé de points de contrôle, la méthode perd de son efficacité.

On remarquera que dans "FORTBUG", toute instruction d'un programme est un point de contrôle. A certains de ces points de contrôle seulement sont rattachées des fonctions ; on peut alors dire qu'ils sont actifs.

IV-2 - ENVIRONNEMENT DU SYSTEME DE MISE AU POINT

Pour le fonctionnement des outils proposés, on a deux types d'informations :

- Celles du premier type décrivent les objets manipulés dans les modules (ensemble des identificateurs avec leurs caractéristiques, structuration du programme, description de l'utilisation de variables particulières et conservation de la trace de l'optimisation).
  Elles sont obtenues lors de la compilation de chaque module.

- Celles du deuxième type décrivent l'implantation en mémoire du système en construction. Elles sont obtenues lors du chargement.

Pourquoi avoir stocké cette information et ne pas avoir adopté un moyen terme consistant à conserver une partie et à recalculer l'autre partie ?

On peut invoquer pour ce choix les raisons suivantes :

. Une meilleure clarté dans l'agencement global des outils d'aide à la programmation (on sait que tel outil accomplit telle fonction et fournit tel résultat).

. On évite de recalculer deux fois la même information (en effet, on peut se rendre compte que tout ce qui est nécessaire pour la phase de mise au point est disponible durant les phases de compilation et de chargement).

. On particularise les modules objet issus d'un programme source écrit en GSL, en leur attachant une description des objets qui y sont manipulés.

A cette méthode (conserver les informations) ainsi qu'à celle du contrôle de l'exécution, on peut faire les reproches suivants :

. Nécessiter l'aide d'un système de gestion de fichiers pour conserver cette information.

. Nécessiter une coopération étroite avec le système d'exploitation, afin de trier les interruptions produites lors de l'exécution du système en

construction (déterminer si une interruption donnée appartient réellement au système en construction ou si elle a été induite par le système de mise au point).

Les outils proposés ont besoin pour fonctionner d'un support. Le premier point en particulier doit donner matière à des développements.

IV-3 - QUELQUES PROBLEMES ET LES LIMITES DES OUTILS

La réalisation des outils proposés s'est heurtée à de nombreuses difficultés ; certaines n'ont pu être que partiellement résolues, soit parce qu'il n'existe pas de solution si l'on veut respecter les caractéristiques fixées pour la construction des outils proposés, soit parce que les solutions possibles sont inacceptables (leurs réalisations présentent un coût trop élevé).

IV-3-1 - Problèmes soulevés par la gestion des registres

Le compilateur GSL réalise une optimisation au niveau des registres dont il a la gestion [C1]. Il est ainsi capable de réutiliser une valeur contenue dans un registre (valeur d'une variable simple ou indicée, d'un indice de tableau, d'une constante, etc...). Des optimisations plus fines [S3] ne sont pas concevables en un seul passage.

Le problème peut se résumer en une ligne : il n'y a plus correspondance biunivoque entre une instruction source et l'ensemble des instructions machine qui en sont issues.

Définissons quelques termes afin d'expliquer le problème dans notre cas particulier.

Pour un module donné, nous avons :

- INS l'ensemble des instructions sources.

- REG l'ensemble des registres disponibles sur la machine (16 dans dans notre cas particulier).

- i∈INS on appelle $UREG_i$ l'ensemble des registres dont l'utilisateur assure la gestion (ce sont soit les registres paramètres, soit les registres qu'il alloue par l'instruction "ALLOCATE"). Cet ensemble est relativement constant et ne varie que lors de la rencontre d'une instruction "ALLOCATE" ou "FREE" lors de la compilation [B5].

- i∈INS on définit $OREG_i$ l'ensemble des registres contenant une valeur et susceptible d'être réutilisée dans la génération du code objet sans modification. Cet ensemble varie lentement lorsque i croit, est détruit lorsqu'on rencontre une étiquette ou certains appels de procédure. De plus, il varie aussi lors de la génération de chaque instruction machine objet correspondant à une instruction source.

- Enfin, pour chaque module, appellons SYSREG les registres qui sont bloqués. Ce sont par exemple les registres qui servent à l'adressabilité ou des registres globaux du système en construction. Cet ensemble est constant pour tout le programme.

L'ensemble des registres significatifs est alors pour chaque instruction i∈INS égal à $UREG_i$ U $OREG_i$ U SYSREG.

Exemple 1 : *Exemple d'optimisation sur un programme.*

Pour chaque instruction GSL, on donne son expansion en langage machine. Les registres sont désignés par leur numéro.

i € INS représente le numéro de l'instruction courante.

```
OPT : PROCEDURE (R1 REG(1) FIXED)
OPTIONS ($DATA REG(12), $PROG REG(11))

ENV (R2 REG(2) PTR) ;
```

Le registre 1 est paramètre, le registre 2 est bloqué, le registre 14 contient l'adresse de retour vers la procédure appelante. Les registres 11 et 12 servent à l'adressage de la procédure.

A ce niveau, pour i=2, on a :

- $UREG_2 = \{1\}$
- $OREG_2 = \emptyset$
- $SYSREG = \{2, 11, 12, 14\}$ *constant par la suite*

```
DCL I FIXED ;
DCL P POINTER ;
DCL Q POINTER EXTERNAL ;
DCL A(10) FIXED BASED(P) ;
DCL B(10) FIXED BASED(Q) ;
```

L'adresse de A est contenue dans la variable P, celle de B dans la variable Q ; ce dernier se trouve dans un autre module.

A ce niveau, pour i=10, on a :

- $UREG_{10} = \{1\}$
- $OREG_{10} = \emptyset$

```
B(I) = 6 ;
   L       15,I
   BCTR    15,0
   AR      15,15
   L       13,=V(Q)
   L       10,0(,13)
   LA      9,6
   ST      9,B(15,10)
```

A ce niveau, pour i=11, on a : $\begin{cases} UREG_{11} = \{1\} \\ OREG_{11} = \{9,10,13,15\} \end{cases}$

```
A(I) = I + B(I) ;
   L     8,P
   A     9,I
   ST    9,A(15,8)
```

A ce niveau, i=12, on a : $\begin{cases} UREG_{12} = \{1\} \\ OREG_{12} = \{8,9,10,13,15\} \end{cases}$

Il est à remarquer que la signification du registre 9 a changé : précédemment, il contenait la valeur de B(I), maintenant il contient la valeur de A(I).

```
ALLOCATE R4 ;
```

A ce niveau, pour i=13, on a : $\begin{cases} UREG_{13} = \{1, 4\} \\ OREG_{13} = OREG_{12} \end{cases}$

```
A(I) = (I = I+1) ;
   LA    7,1
   A     7,I
   ST    7,I
   ST    7,A(15,8)
```

A ce niveau, pour i=14, on a : $\begin{cases} UREG_{14} = \{1, 4\} \\ OREG_{14} = \{7,8,10,13\} \end{cases}$

```
END ;
   BR    14
```

Tous les registres sont libérés, sauf ceux appartenant à la liste de retour ou à l'environnement.

Signification du contenu des registres appartenant à $OREG_i$, pour l'exemple 1 :

(i ∈ INS représente le numéro de l'instruction courante).

. *au niveau de l'instruction i = 11 :*

le registre 9 contient la valeur de B(I)
le registre 10 contient la valeur de la variable Q
le registre 13 contient l'adresse de la variable Q
le registre 15 contient la valeur de I multipliée par 4

. *au niveau de l'instruction i = 12 :*

la signification des registres 10, 13 et 15 est inchangée
le registre 8 contient la valeur de la variable P
le registre 9 contient la valeur de A(I)

. *au niveau de l'instruction i = 14 :*

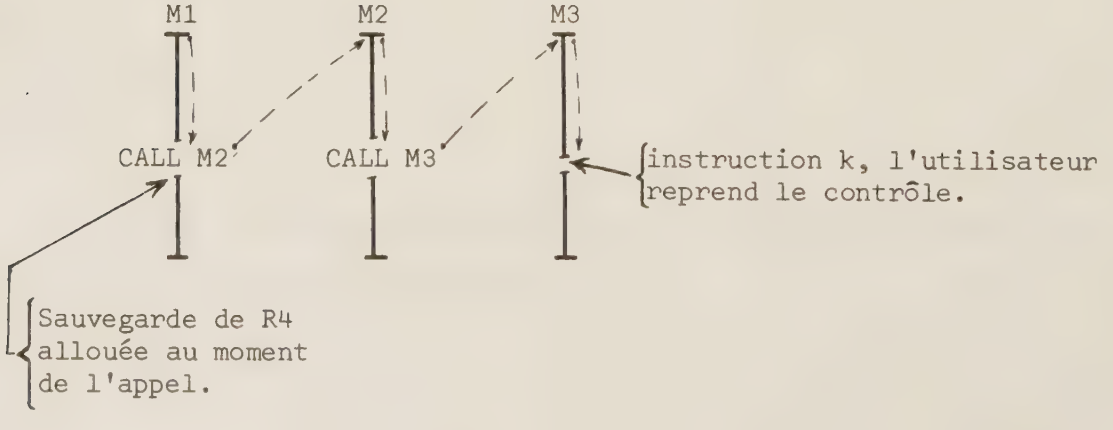
le registre 7 contient la nouvelle valeur de la variable I
le registre 8 contient la valeur de la variable P
le registre 10 contient la valeur de la variable Q
le registre 13 contient l'adresse de la variable Q

Les problèmes posés sont les suivants :

a) Lors de la modification d'une variable, il faut aussi modifier les registres qui contiennent la valeur de cette variable directement ou indirectement lorsque deux variables occupent la même zone (ce sont les registres de $OREG_i$).

Lors de la modification d'un registre, il faut s'assurer que l'utilisateur en a la gestion au point où le contrôle lui a été rendu (qu'il appartient bien à l'ensemble $UREG_i$).

Cette même gestion des registres soulève le problème suivant : supposons que nous ayons trois modules M1, M2 et M3. M1 active M2 ; M2 à son tour active M3 (cf schéma 1). L'utilisateur reprend le contrôle sur l'instruction k du module M3. Lors de l'appel de M2 dans M1, le registre R4 par exemple était alloué ; pour en connaître sa valeur, l'utilisateur qualifie le module M1 et demande à affichier R4. Tout ceci est correct ; il faut retrouver où le registre R4 a été sauvegardé au moment de l'appel. Ce problème est simple lorsque les conventions de liaison sont standards ; en GSL, elles sont entièrement paramétrées et dépendent des procédures appelées et appelantes. L'utilisateur peut s'il le veut gérer lui-même les sauvegardes (exemple 2). Il existe des solutions à ce problème : soit conserver dans la pile de contrôle une trace décrivant la sauvegarde pour chaque module actif, soit reconstruire "l'histoire" de la sauvegarde à partir des descripteurs de procédures. Dans tous les cas, les solutions sont coûteuses et parfois n'assurent pas une entière fiabilité.



- Schéma 1 -

Exemple 2

Nous donnons ici un exemple simplifié de conventions de liaison entre deux procédures compilées indépendamment.

Cet exemple a été extrait des programmes vérifiant la cohérence des liens lors de l'intégration. La procédure MCH10 calcule la taille du descripteur principal d'un élément ; lorsque ce dernier est un descripteur de procédure, MCH10 appelle MCH11 qui calcule la taille de l'arbre décrivant cette procédure.

Les sauvegardes sont assurées localement dans une pile directement gérée par le programmeur.

La procédure MCH10 sauvegarde les registres de la procédure qui l'appelle (ENV($SAVAREA...)) et possède elle-même une zone de sauvegarde (OPTIONS($SAVAREA)); lors de l'appel de MCH11, le registre 4 contient l'adresse de la pile des sauvegardes.

Dans la procédure MCH11, aucune variable locale est générée ; la zone de sauvegarde est repérée par le registre 4 ; lors de l'appel récursif de MCH11 seul, les registres significatifs et susceptibles d'être changés sont sauvés sur le sommet de la pile.

```
MCH10 : PROC (RETURNS(REG(15)FIXED))
ENV($SAVAREA, BASE REG(10) PTR)
OPTIONS($SAVAREA) ;
      |
      |
DCL RSAV4 REG(4) PTR ;
DCL 1 COMMUN BASED(BASE),
    2 DATASAVE PTR,      DATASAVE contient l'adresse de la pile de sauvegarde.
    2 LGTHSAVE FIXED,    LGTHSAVE contient la taille d'une entrée dans la pile de sauvegarde.
    2 DESCR PTR ;
DCL MCH11 PROC
ENV($ENTADDR REG(15), REG(10)PTR, REG(3) PTR, REG(4) PTR) EXT ;
      |
      |
      |
      |
ALLOCATE RSAV4 ;
RSAV4 = DATASAVE ;      RSAV4 contient l'adresse de la pile de sauvegarde.
CALL MCH11 ;            Lors de l'appel, les registres sont sauvegardés dans la pile.
FREE RSAV4 ;
      |
      |
   END ;


MCH11 : PROC
ENV($ENTADDR REG(15), BASE REG(10) PTR, RPTR3 REG(3) PTR, RSAV4 REG(4) PTR)
OPTIONS($REENTRANT, $SAVE REG(4)) ;
      |
      |
DCL 1 COMMUN BASED(BASE),
    2 * PTR,
    2 LGTHSAVE  FIXED,
    2 * FIX ;
      |
      |
RSAV4 = RSAV4 + LGTHSAVE ;  Le niveau de la pile est mis à jour avant l'appel.
CALL MCH11 ;
RSAV4 = RSAV4 - LGTHSAVE ;  Le niveau de la pile est remis à jour après l'appel.
      |
      |
END ;
```

b) Lors d'un arrêt anormal, c'est-à-dire dû à une erreur du programme qui s'exécute, la gestion des registres pose le problème suivant :

Supposons que l'arrêt se soit produit entre l'instruction i et i+1 ; on ne connaît que l'état des registres sur l'instruction i et sur l'instruction i+1. Entre les deux, lors de la génération du code objet, on n'a pas conservé la trace de l'évolution des registres.

Dans ce cas, on a pris l'option de travailler avec l'état des registres relatifs à l'instruction i (c'est-à-dire satisfaire les requêtes et reprendre l'exécution). Cette option peut engendrer dans certains cas des effets de bord ou ne pas être totalement fiable (cf exemple 3).

Le fait de travailler avec l'état des registres relatifs à l'instruction i+1 n'est pas plus fiable.

Garder l'état des registres pour chaque instruction du code objet s'avère la seule solution fiable, mais d'un coût élevé.

Exemple 3

Reprenons les déclarations données dans l'exemple 1 ; supposons que nous possédions de par l'optimisation la valeur de I dans le registre 15.

$OREG_k = \{15\}$

```
k       A(I) = B(I=I+1) ;
k1        BCTR   15,0
k2        AR     15,15
k3        AR     15,15
k4        LA     13,1
k5        A      13,I
k6        ST     13,I
k7        BCTR   13,0
k8        AR     13,13
k9        AR     13,13
k10       L      10,=V(Q)
k11       L      9,0(,10)
k12       L      8,P
k13       L      7,B(13,8)
k14       ST     7,A(15,9)
```

$OREG_{k+1} = \{7,8,9,10,13\}$

Signification des registres (exemple 4) :

au niveau de l'instruction k :

le registre 15 contient la valeur de la variable I

au niveau de l'instruction k+1 :

le registre 7 contient la valeur de la variable B(I)
le registre 8 contient la valeur de la variable P
le registre 9 contient la valeur de la variable Q
le registre 10 contient l'adresse de la variable Q
le registre 13 contient la valeur I multipliée par 4

au niveau de l'instruction assembleur k_{11} avant qu'elle ne soit exécutée :

le registre 10 contient l'adresse de la variable Q
le registre 13 contient la valeur I multipliée par 4
le registre 15 contient le déplacement de l'élément A(I)
I possédant son ancienne valeur.

Supposons que la valeur du registre 10, obtenue après son chargement, soit erronée et provoque un arrêt anormal sur l'instruction assembleur k11. On se rend aisément compte des effets de bord qu'il est possible d'engendrer en se plaçant en k ou en k+1.

c) Pourquoi ne pas avoir permis la reprise de l'exécution par un "GOTO" ?

Les problèmes que pose l'exécution d'un "GOTO" apparaissent fort bien dans la description du fonctionnement d'un interpréteur [B4] [B6].

En GSL, un "GOTO" à l'intérieur d'un même module nécessite :

- une vérification sur la validité du "GOTO" par rapport à la structure statique de bloc.
- une vérification et une mise à jour des valeurs des registres servant à l'optimisation (ensemble $OREG_i$).
- une vérification sur les registres dont la gestion est assurée par l'utilisateur.

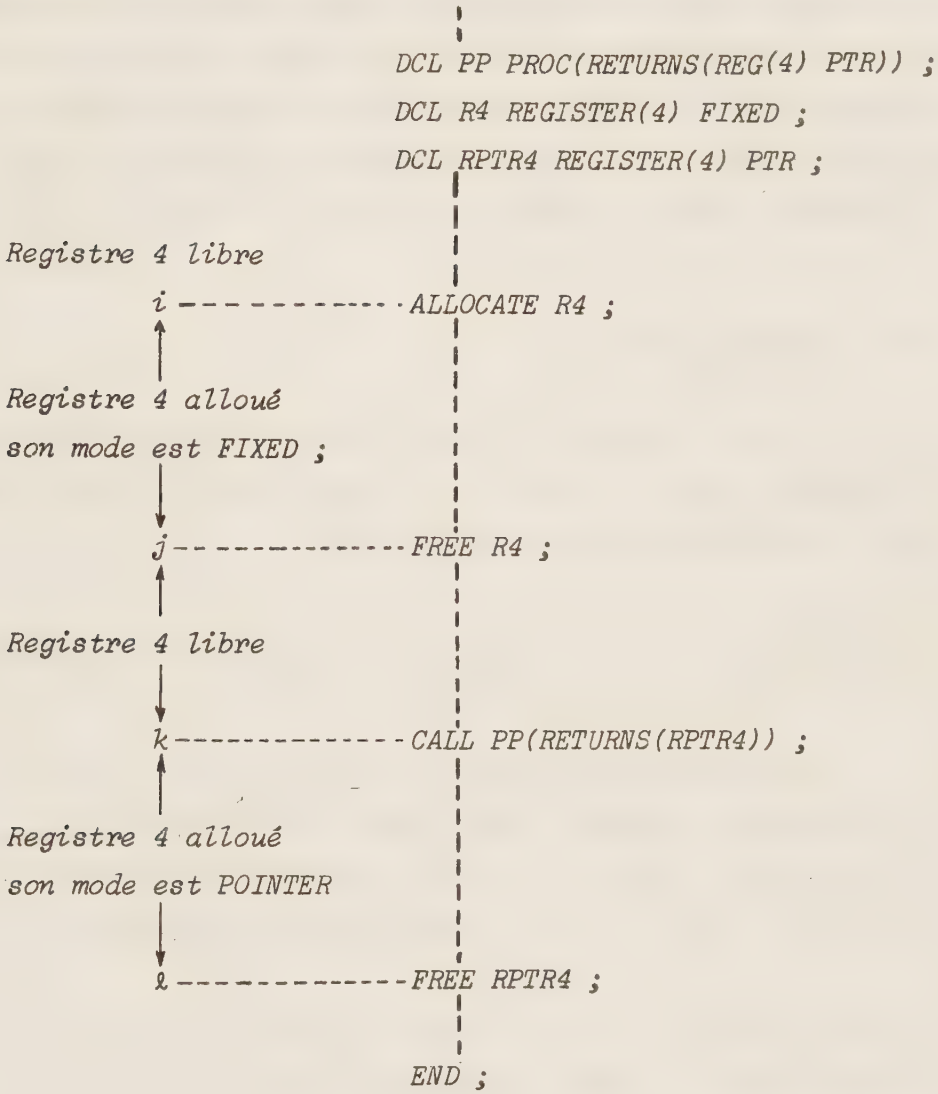
Si l'on reprend l'exemple 1, un "GOTO" de l'instruction 14 vers l'instruction 11 nécessite la remise à jour du registre 9, les registres 10, 13 et 15 restant inchangés.

Si l'on prend l'exemple 4, un "GOTO" de la portion de programme [kℓ] vers la portion [ij] doit être interdit, le registre alloué n'ayant pas la même signification.

Par l'intermédiaire de la combinaison de la requête "QUALIFY" et "GOTO", on pourrait abandonner l'exécution de plusieurs modules actifs ; on aurait alors, s'ajoutant aux précédents, les problèmes suivants :

- Vérifier si le module est actif.
- Restaurer l'état de la pile de contrôle.
- Restaurer les registres sauvegardés.
- Vérifier l'état des registres servant à passer des résultats en retour.

Exemple : *Vérifications de la validité des registres alloués*



Si l'on veut faire un GOTO de l'instruction m vers l'instruction n, il est nécessaire, afin de diminuer les risques d'erreurs, que l'ensemble des registres alloués à l'instruction n soit inclus dans l'ensemble des registres alloués à l'instruction m ; de plus, les registres identiques doivent être de même mode.

Ainsi : - un GOTO de [ij] vers]kℓ] est rejeté le registre 4 a des modes différents

- un GOTO de [ij] ou de [kℓ] vers]j,k[est accepté

- un GOTO de]j,k[vers]ij] ou]k,ℓ] est rejeté.

d) Dans cette version du système d'aide à la mise au point, l'accès aux variables "dynamiques" (Registres), automatiquement sauvegardées lors d'un appel, n'est pas autorisé. Pour cette raison, la commande "QUALIFY" n'a comme paramètre qu'un nom de module. Si l'accès avait été permis, on aurait été obligé de rajouter le niveau de récursivité du module afin de désigner le groupe de variables à atteindre.

IV-3-2 - Restrictions

Lors de cette étude, nous avons été obligés de nous accommoder d'un certain nombre de limites ; ces dernières paraissent s'appliquer à tout système semblable à celui proposé.

a) Les vérifications effectuées dépendent totalement du langage. Elles sont limitées par les renseignements fournis dans le mode des éléments.

Un "hardware adéquat" ou une connaissance plus précise du système objet pourraient pallier à cette limite. Des exemples de ces limites sont les erreurs dues à une activation de procédure (on ne vérifie que la cohérence de l'appel) ou à des manipulations d'adresses.

b) On a vu qu'une pile (pile de contrôle) mise à jour à l'activation et désactivation de chaque module nous permet de connaître l'ensemble des modules actifs.

Or, la validité du contenu de la pile repose sur l'hypothèse suivante : tout appel est procédural (ou fermé).

Dans un système d'exploitation, il est fréquent que plusieurs groupes de modules s'exécutent en "parallèle", la synchronisation étant assurée par un "Dispatcher". Dans ce cas, l'hypothèse précédente est invalidée.

Une solution consisterait à déterminer des groupes de modules dans lesquels l'hypothèse serait vérifiée et à affecter une pile de contrôle par groupe. Par suite de la possibilité de passage de paramètres de type procédure, on ne peut pas déterminer ces groupes avant l'exécution.

c) La masse d'informations à gérer peut représenter un handicap. En effet, si l'on considère les fonctions que l'on veut posséder, elles sont identiques à celles fournies par un compilateur générateur de tests. La différence vient de ce que dans un compilateur générateur de tests, on sait exactement ce qu'il faut garder comme information à la compilation, alors que dans notre cas, on ne peut pas savoir exactement ce qui sera nécessaire.

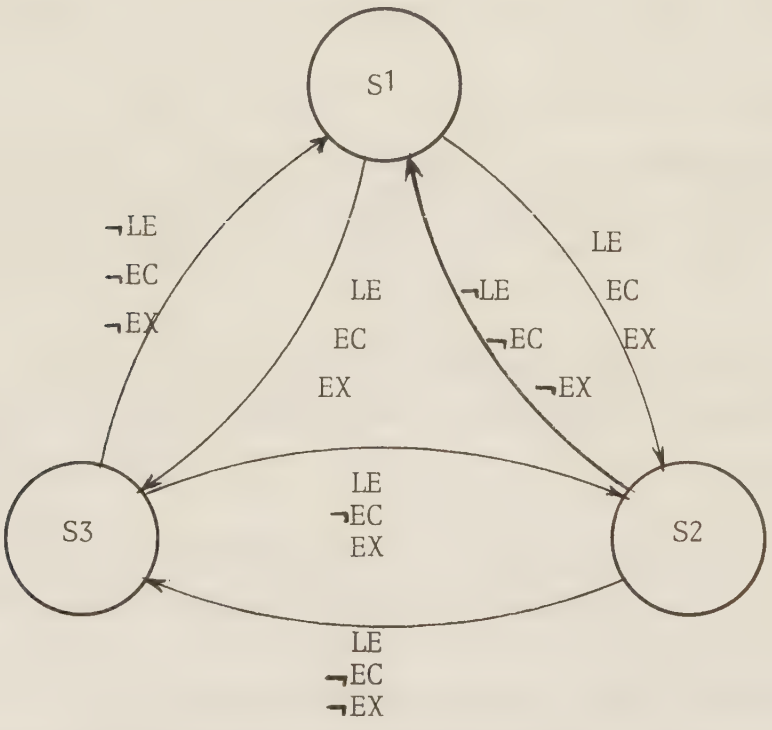
IV-3-3 - Extensions "Hardware"

a) Mécanisme de protection

Sur le modèle 67 de la série des calculateurs IBM S/360, il existe un mécanisme de protection en lecture/écriture par clef. Ce mécanism est difficilement utilisable pour protéger le système de mise au point des erreurs du système en construction lorsque ce dernier peut manipuler les clefs de protection.

Possédant un mécanisme de protection plus élaboré, il serait intéressant de réaliser la protection définie par le schéma 2.

Schéma 2 - Protection entre le système de mise au point et le système testé



Les flèches indiquent le sens de l'accès.

EC accès en écriture autorisé (¬ EC interdit)

LE accès en lecture autorisé (¬ LE interdit)

EX accès en exécution autorisé (¬ EX interdit)

S1 système d'aide à la mise au point

S2 fonctions d'observation écrites par l'utilisateur

S3 système en construction.

- Schéma 2 -

b) Contrôle de l'adressage

Il est intéressant, lors de la mise au point, de savoir quand et par qui une zone donnée est référencée ; un tel contrôle n'est possible qu'avec un "hardware" adéquat.

En effet, on peut simuler l'exécution du programme en construction, mais ceci est extrêmement lent.

Cette fonction est en général disponible à partir du pupître des calculateurs ; un effort a été fait dans la série des calculateurs IBM S/370 [I9] où l'accès à cette fonction peut être fait par programme. On peut ainsi positionner un arrêt et reprendre le contrôle lorsqu'un branchement est réalisé avec succès, qu'un registre est modifié, une instruction exécutée ou lorsqu'une partie de mémoire centrale est atteinte.

IV-4 - SYSTEME D'AIDE A LA PROGRAMMATION

a) On possède une masse d'informations extrêmement importante décrivant les modules du système en construction. Il serait donc intéressant de ne pas limiter son exploitation aux deux outils présentés.

Si l'on observe un programmeur travailler à la création ou à la maintenance de programmes, on s'aperçoit qu'une grande partie de son temps est employée à des manipulations de listes de programmes, à rechercher où une variable globale est utilisée, quel est le mode d'une autre, quels sont les paramètres d'une procédure, où une procédure est appelée ...

La fonction de vérification du chargeur assisté fournit le graphe statique des appels des modules composant le système en construction ainsi que les points d'utilisation des variables globales. Pour chaque module, on connaît les attributs, les noms, les utilisations, etc ... des objets apparaissant dans le module.

Par l'intermédiaire d'un système d'interrogation, le programmeur pourrait enquêter sur l'état de ces relations, même pendant la phase de mise au point.

Il pourrait alors demander où la variable "I" par exemple est déclarée, dans quels modules elle est utilisée, dans quelles instructions elle est directement modifiée, quel est l'ensemble des modules appelant un module d'un nom donné, etc ...

b) Nous nous servons actuellement du système de gestion de fichier disponible sous CMS ; il existe entre les fichiers décrivant un système en construction des relations ; le système CMS ne permet pas d'en tenir compte.

Considérons un système en construction ; pour chaque module, on possède trois fichiers (le fichier source, le fichier contenant le code objet, le fichier contenant les descriptions) ; lors de l'intégration, un fichier contenant la description de l'implantation du système en mémoire est créé. On possède alors un lien entre les divers triplets

décrivant le système en construction.

La modification du contenu d'un des fichiers source peut fausser l'interprétation du contenu des autres fichiers. Un système de gestion de fichier capable de détecter la coexistence, pour un même module, des fichiers de versions différentes s'avérerait utile.

Dans un deuxième temps, il serait intéressant d'envisager le travail simultané de plusieurs personnes sur un même système. Cette méthode n'est pas unanimement appréciée,surtout lorsque "les modifications du voisin" viennent perturber le fonctionnement de la partie que l'on construit.

Cependant, elle permet d'accélérer la construction d'un système et d'en faciliter la maintenance.

En effet, supposons que l'on ait une version d'un système au point ; on désire lui apporter des améliorations ou changements. Si le découpage du système a été fait correctement lors de sa construction, il ne s'agit que de modifier quelques fonctions.

Affectons une personne à la modification de chaque fonction par exemple ; chacune d'elle se trouvera en présence de l'ensemble des modules du système initial et de la fonction dont elle a la charge.

Un tel système d'aide à la construction de système nécessite beaucoup plus qu'une simple gestion de fichiers ; il nécessite un "moniteur" capable de gérer les différents niveaux de versions des fichiers décrivant un module (on peut ajouter aux fichiers déjà cités la description fonctionnelle du module, son organigramme etc ...) et d'en assurer la protection (accès et modifications sélectives).

C O N C L U S I O N

La conception et la réalisation de ce prototype nous ont amenés à un certain nombre de réflexions relatives aux outils destinés à faciliter la construction de systèmes.

Les programmes composants des outils proposés ont été écrits en langage GSL. Ceci nous a montré que l'utilisation d'un langage de haut niveau dans la construction d'un système était une amélioration notoire. Cependant, si l'on voulait conserver les avantages acquis, il était nécessaire de posséder des outils appropriés, pour assister chacune des phases ultérieures.

Par exemple, si l'on veut se dégager des inconvénients créés par la manipulation d'adresse et de code machine, l'utilisation d'outils d'aide à la programmation ne doit pas impliquer la connaissance des caractéristiques du code objet engendré par le compilateur.

On est alors amené à constater :

- qu'une compilation vers du code exécutable s'accompagne d'une dégradation de l'information contenue dans le code source.
- que cette information, extraite du texte source par le compilateur pour engendrer le code objet, est perdue après la compilation.
- que le fonctionnement des outils d'aide à la programmation nécessite la connaissance de cette même information.

Une décompilation du code objet est quasiment impossible ; de plus, suivant les compilateurs, on ne retrouvera qu'un programme équivalent au programme original.

Si l'on veut posséder des outils efficaces et pallier à cette dégradation d'information, la conception et l'élaboration de ces outils

doivent avoir lieu en même temps que celles du langage et du compilateur. Les fonctions supplémentaires à inclure dans le compilateur sont alors bien définies et réalisées de façon propre. Cependant, il est à déplorer que la construction d'outils d'aide à la programmation n'est pas le principal souci des constructeurs de "Software" [G5].

L'application d'une telle idée donnerait naissance à des systèmes de programmation où le compilateur et les outils formeraient un tout ; le langage de programmation fournirait l'unité du système. Ceci éviterait de limiter le rôle des compilateurs à celui d'analyseur et de traducteur.

Des tentatives d'unification ont eu lieu dans le but de posséder un ensemble unique de commande de mise au point par exemple, quel que soit le langage [B2].

On voit deux objectifs majeurs à ce type d'unification :

- Des langages de niveaux ou d'orientations différents ne manipulent pas les mêmes objets de la même manière et ne présentent pas des aspects identiques.

- Cette unification n'est qu'une enveloppe des problèmes fondamentaux de contrôle et d'identification.

Les deux outils présentés -tout particulièrement la partie ayant trait aux vérifications du chargeur assisté en fonction depuis assez longtemps- nous ont permis de faire les constatations suivantes :

- Leur emploi diminue le travail qui incombe au programmeur et évite parfois la recherche fastidieuse d'erreurs.

- Les incohérences de liaisons entre modules sont souvent dues à une méconnaissance du langage (Par exemple, les options prises par défaut).

- Le programmeur tend à apporter moins de soins aux vérifications qu'il fait manuellement ; il se repose en effet sur les outils qu'il possède pour détecter les erreurs.

- Les outils d'aide à la mise au point, quel que soit leur degré de perfectionnement, n'offrent qu'un moyen d'investigation plus aisé. Toutefois, pour un programmeur expérimenté,la contribution apportée par un outil d'aide à la mise au point dans la recherche des erreurs(en particulier des erreurs "d'algorithme") est loin d'être négligeable.

- La définition du langage possède une part importante dans les vérifications effectuées par de tels outils.

- Le contrôle interactif de l'exécution apporte une solution souple, lors de la recherche d'une erreur dans un algorithme.

- Les problèmes de l'identification et du contrôle de l'exécution sont les points fondamentaux des outils d'aide à la détection des erreurs.

Si l'on veut faire un pas de plus dans le domaine de l'aide à la programmation, il semble que la constitution d'un système de gestion automatique d'un projet et l'intégration du compilateur et des outils dans ce système offrent des développements nouveaux.

Il paraît en effet plus aisé et fiable d'assurer des opérations telles que : l'élaboration d'une nouvelle version d'un système, sa mise à jour, ses tests, etc... de manière automatique plutôt que manuellement.

ANNEXE A

RAPPEL SUR LES GRAMMAIRES DE TYPE 3 ET SCHEMA DE L'ANALYSE

A chaque commande est associée une grammaire; pour chacun des états on indique l'ensemble des états vers lesquels on peut transiter les éléments permettant d'effectuer le transit et les fonctions sémantiques que l'on doit appeler lors des transitions.

Exemple :

Grammaire : $G = \{V_T, V_N, E1, P\}$

éléments terminaux $V_T = \{id, =, cst, +, -, \dashv\}$

éléments non terminaux $V_N = \{E1, E2, E3, E4, E5\}$

L'ensemble des règles P :

$E1 \rightarrow id\ E2$

$E2 \rightarrow =\ E3$

$E3 \rightarrow id\ E4 \mid$
$\quad cst\ E4$

$E4 \rightarrow +\ E5 \mid$
$\quad -\ E5 \mid$
$\quad \dashv$

$E5 \rightarrow id\ E4 \mid$
$\quad cst\ E4$

Nous possédons ici une grammaire de type 3 qui permet d'engendrer les expressions :

$$id = \{id \mid cst\} \quad \{\{+ \mid -\}\ \{id \mid cst\}\}^* \dashv$$

{ } sert à déterminer un groupe.

| détermine un choix, exemple : {id | cst} id ou cst.

L'automate correspondant à cette grammaire est le suivant :

. Etats de l'automate : E = {E1, E2, E3, E4, E5, E6}

. Etat initial E1

. Etat final E6

. Ensemble des symboles terminaux V_T = {id, cst, =, +, -, ⊣}

. Fonction de transition $\delta : E \times V_T \rightarrow E$

δ(E1, id) = E2

δ(E2, =) = E3

δ(E3, id) = E4

δ(E3, cst) = E4

δ(E4, +) = E5

δ(E4, -) = E5

δ(E4, ⊣) = E6

δ(E5, id) = E4

δ(E5, cst) = E4

La présentation graphique de l'automate est :

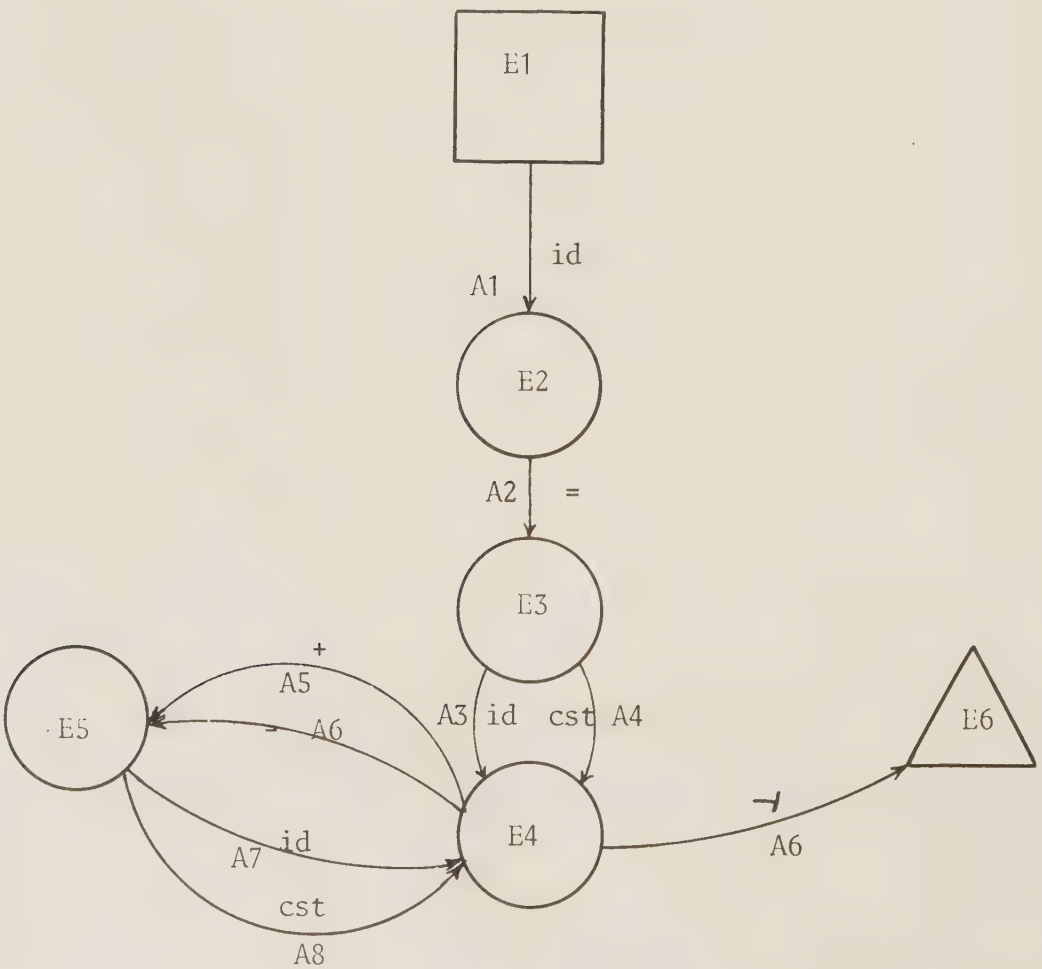
Convention :

état initial

état intermédiaire

état final.

Pour chacun des arcs du graphe, on spécifie le symbole de l'ensemble V_T qui permet d'opérer la transition, ainsi que la ou les fonctions sémantiques à appeler si la transition a lieu.



La représentation d'un tel graphe est aisée, et permet d'associer les fonctions sémantiques à appeler.

Type f signifie que l'état est final
Type i signifie que l'état est initial
Type t signifie que l'état est transitoire.

état	type de l'état	unité syntaxique	fonction	états suivants
E1	i	id	A1	E2
E2	t	=	A2	E3
E3	t	id cst	A3 A4	E4
E4	t	+ - ⊣	A5 A6 A9	E5 E5 E6
E5	t	id cst	A7 A8	E4 E4
E6	F			

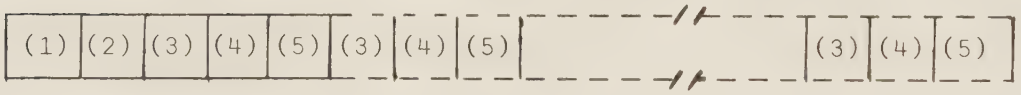
Fonctionnement de l'analyse (cf. figure 2) :

Un préprocesseur isole les unités syntaxiques terminales apparues dans la chaîne d'entrée et leur assigne un numéro.

Un interpréteur analyse syntaxiquement la chaîne codée fournie par le préprocesseur à l'aide de la grammaire (cf. figure 1). De plus, l'interpréteur appelle les fonctions sémantiques nécessaires à l'exécution de la commande.

Les fonctions sémantiques ont deux rôles :
- vérifier la cohérence de la commande par rapport aux informations gardées lors de la compilation.
- assurer l'exécution de la commande.

Figure 1 : Forme d'une entrée dans la table représentant la grammaire.
Il y a une entrée de longueur variable par état.



(1) indique le type de l'état (initial, final ou transitoire)

(2) donne le nombre d'octets occupés par l'entrée

(3) contient le code de l'élément qui permet d'appeler la fonction sémantique dont le numéro est en (4) et d'effectuer la transition vers l'état suivant dont le déplacement est dans (5)

(4) numéro de la fonction sémantique à appeler

(5) déplacement de l'état suivant dans la table.

Le groupe (3) (4) (5) se répète autant de fois qu'il y a d'états successeurs possibles (ou d'alternatives dans la règle).

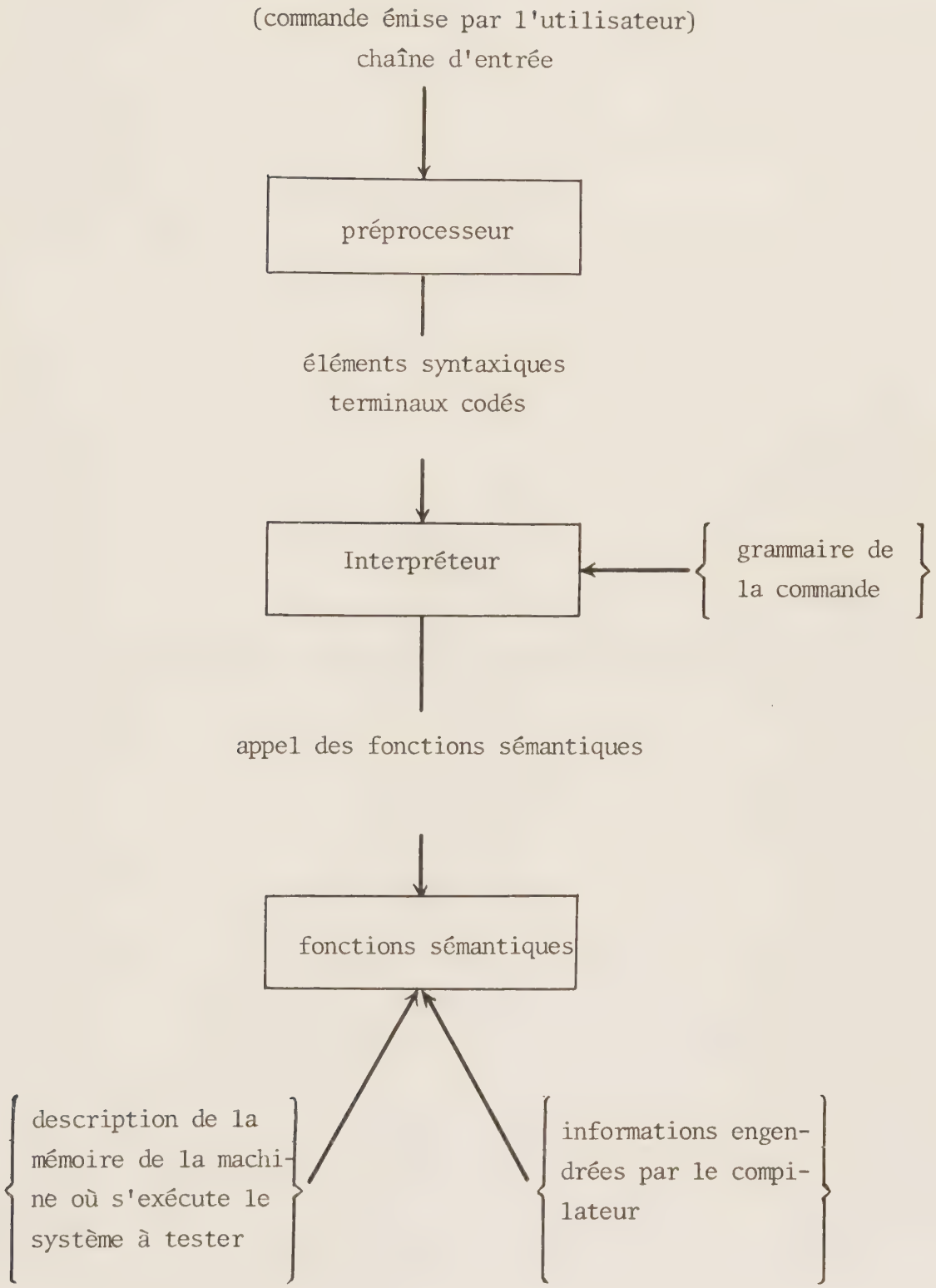


FIGURE 2 : Schéma de l'analyse et de l'exécution d'une commande

ANNEXE B

GRAMMAIRE DES COMMANDES D'AIDE A LA MISE AU POINT

Axiome : <commande>

<commande> → [ADDR] <corps de l'addr>|[QUALIFY]<élément qualifié>|
[QUALIFY]<fin>|[identificateur]<affich modif>|
[constante]<cst qualification>|[AT]<corps de l'arrêt>|
[TRACE]<fin>|[SUBTRACE]<fin>|[QUIT]<fin>|
[RESET]<reset argument>|[QUERY]<corps du query>
[GET]<mac name>|[KILL]<mac name>|
[GO] <fin>|[CALL]<call argmt>

<corps de l'addr> → [( ] <argument de l'addr>

<argument de l'addr> → [identificateur] <reste de l'addr>

<reste de l'addr> → [( ] <indicage d'addr>|[)] <fin>

<indicage d'addr> → [identificateur]<fermeture d'indice>|[constante]
<fermeture d'indice>

<fermeture d'indice> → [)] <fermeture>

<fermeture> → [)] <fin>

<élément qualifié> → [:]<fichier qualifié> [identificateur de procédure
externe]<fin>

<fichier qualifié> → [nom de fichier de type "debug"]<fin>

<afich modif> → [?]<fin>|[=]<modif>|[(]<indicage>|[→]<qualid>

<cst qualification> → [→]<qualid>

<qualid> → [identificateur de type basé]<affich modif>

<indicage> → [constante]<tab ou chaîne>

<tab ou chaîne> → [:]<offset>|[,]<tableau de chaîne>|[)]<fin de subscript

<offset> → [constante]<fin d'offset expr>

<tableau de chaîne> → [constante]<offset expr>

<offset expr> → [:]<offset>

<fin d'ofset expr> → [)]<fin de subscript>

<fin de subscript> → [?]<affich>|[=]<modif>|[→]<qualid>

<affich> → [↵]

<modif> → [constante]<fin>

<fin> → [↵]

<corps d'arrêt> → [:]<spécial arrêt>|<repérage d'inst><action sur arrêt>

<action sur arrêt> → [STOP]<cond mac>|[RUN]<mac seule>

<cond mac> → [MACRO]<mac name>|[IF]<cond expr>|<↵>

<mac name> → [nom de macro]<fin>

<cond expr> → [COUNT]<rest expr>

<rest expr> → [>]<val comp> [=]<val comp>|[<]<val comp>

<val comp> → [constante entière]<fin>

<mac seule> → [MACRO]<mac name>

<spécial arrêt> → [ENTPROC]<exproc name>|[EXITPROC]<exproc name>|
[REFERENCE]<idéréfé>|[SUBSCRIPT]<subscriptuse>|
[ALL]<borne all>

<exproc name> → [identificateur de procédure externe]<exproc name>|
[identificateur de procédure externe]<entsortaction>

<entsort action> → [STOP]<fin>

<idéréfé> → [identificateur autre qu'étiquette et procédure]
<réféaction>

<réféaction> → [STOP]<fin>

<subscriptuse> → [identificateur de tableau ou de chaîne]<fin>

<borne all> → [FROM]<borne un>

<borne un> → <repérage d'inst><borne deux>

<borne deux> → [TO]<repérage d'arrêt><all action>

<all action> → [RUN]<mac seule>|[STOP]<mac all>

<mac all> → [MACRO]<mac name>|[↵]

<repérage d'inst> → [identificateur de type étiquette]|[numéro d'instruction

<reset argument> → [AT]<atall argument>|[ALL]<fonction>|[TRACE]<fin>
[MACRO]<localisation de at>|[COUNT]<localisation de at>|
[STOP]<porteur d'arrêt>|[SUBSCRIPT]<subscriptuse>

<at all argument> → [ALL]<fin>|[ALL]<action du all>

<action du all> → [STOP]<fin>|[MACRO]<fin>

<fonction> → [MACRO]<fin>|[MACRO]<mac name>|[STOP]<fin>|[COUNT]<fin>

<localisation de at> → [AT]<repérage d'instruction><fin>

<porteur d'arrêt> → <localisation de at>|[REFERENCE]<reset réfé>|
[ENT PROC]<liste proc>|[EXIT PROC]<liste proc>

<liste proc> → [nom de procédure externe]<liste proc>|[nom de procédure
externe]<fin>

<reset réfé> → [identificateur autre qu'étiquette ou procédure]<fin>

<corps de query> → [COUNT]<count query>|[MACRO]<mac name>|[STOP]<fin>|
[ENTPROC]<fin>|[EXITPROC]<fin>|[REFERENCE]<fin>|
[AT]<at all query>|[MACRO]<fin>

<count query> → [AT]<repérage d'instruction><fin>|[ONLY]<fin>

<at all query> → [ALL]<fin>

<call argmt> → [nom de procédure GSL.suivie de ses arguments]<fin>

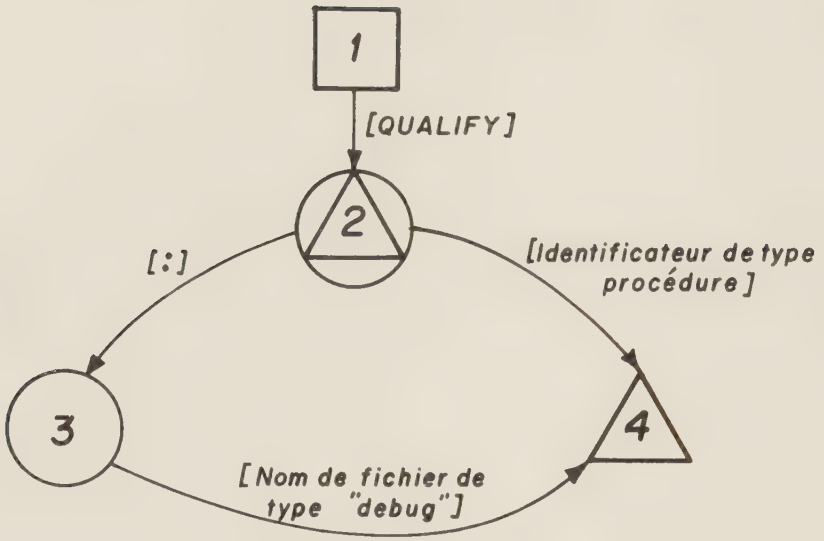
<fin> → [⬅]

Il est à remarquer que cette grammaire de type 3 comporte des règles de la forme A → aBC (avec A, B, et C appartenant à l'ensemble des éléments non terminaux, et 'a' étant un terminal). Ceci est intentionnel, et a pour but de faciliter la compréhension.

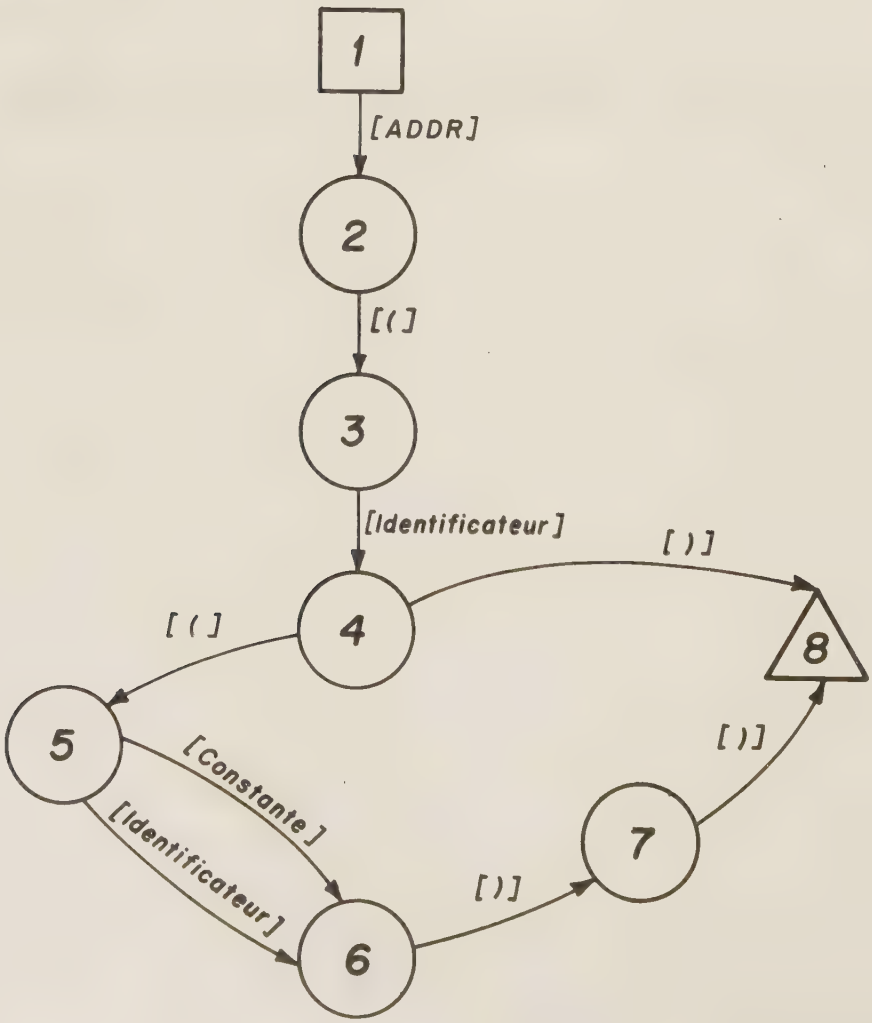
ANNEXE C

AUTOMATES CORRESPONDANT AUX PRINCIPALES COMMANDES

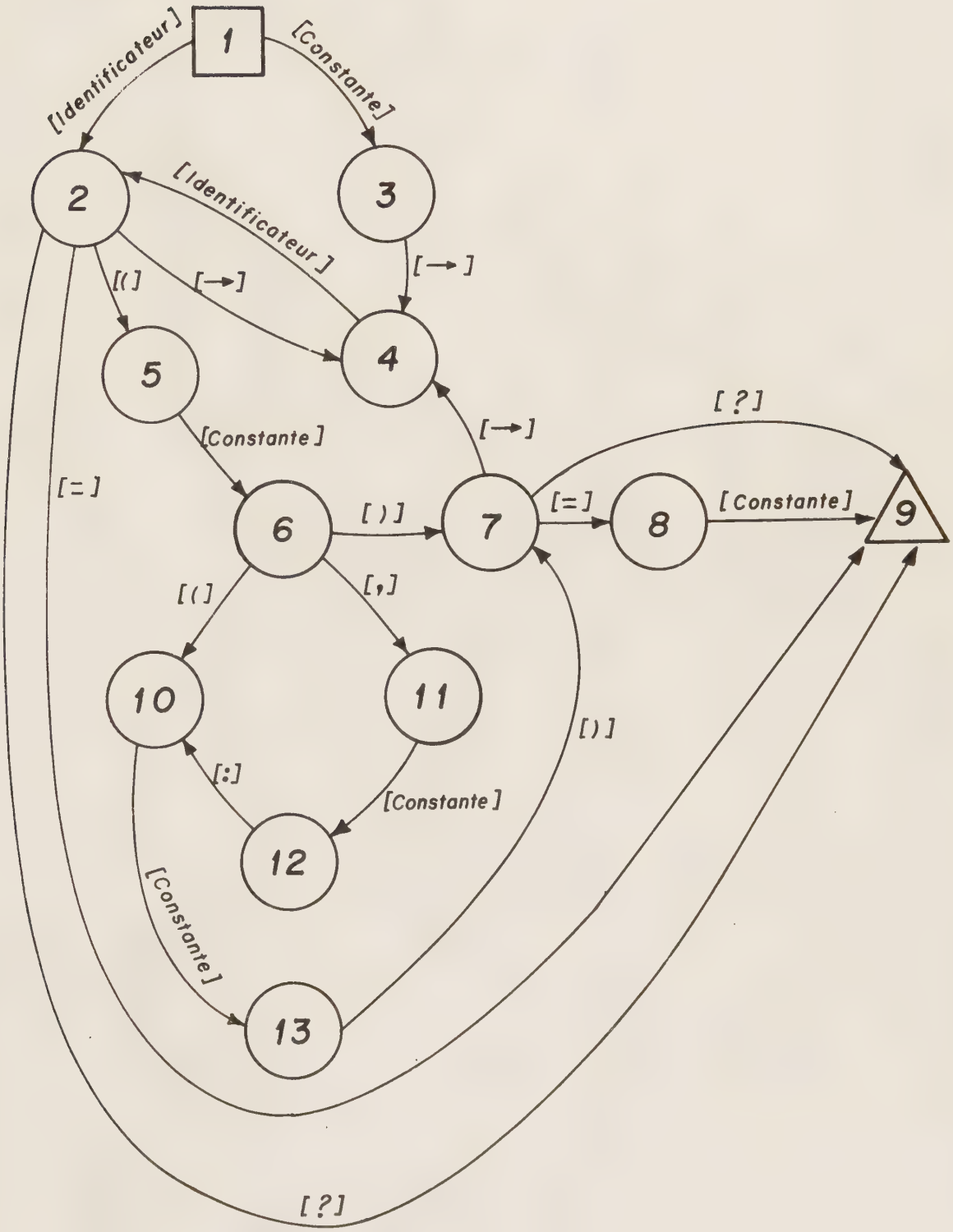
Commande QUALIFY



Commande ADDR

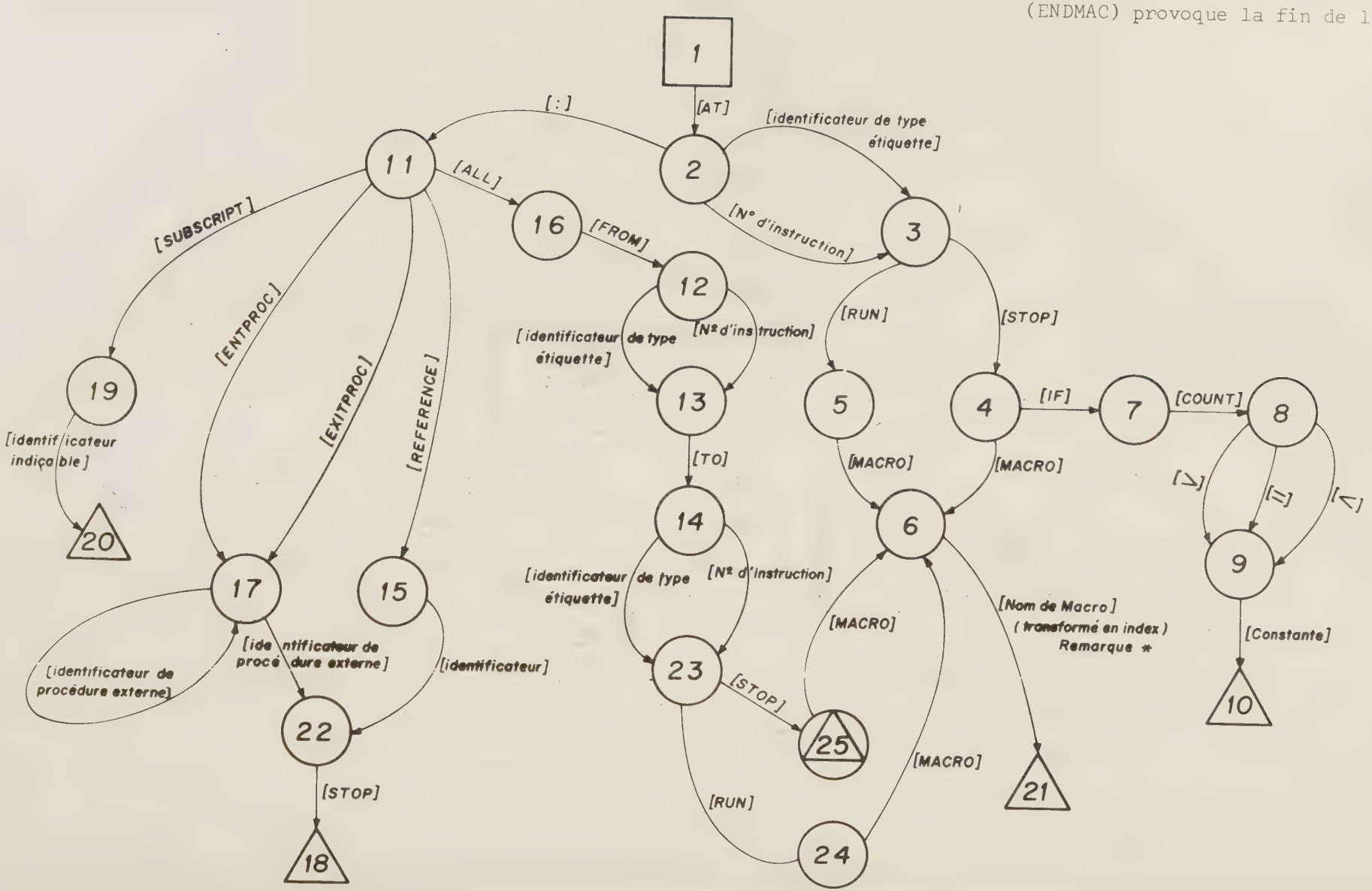


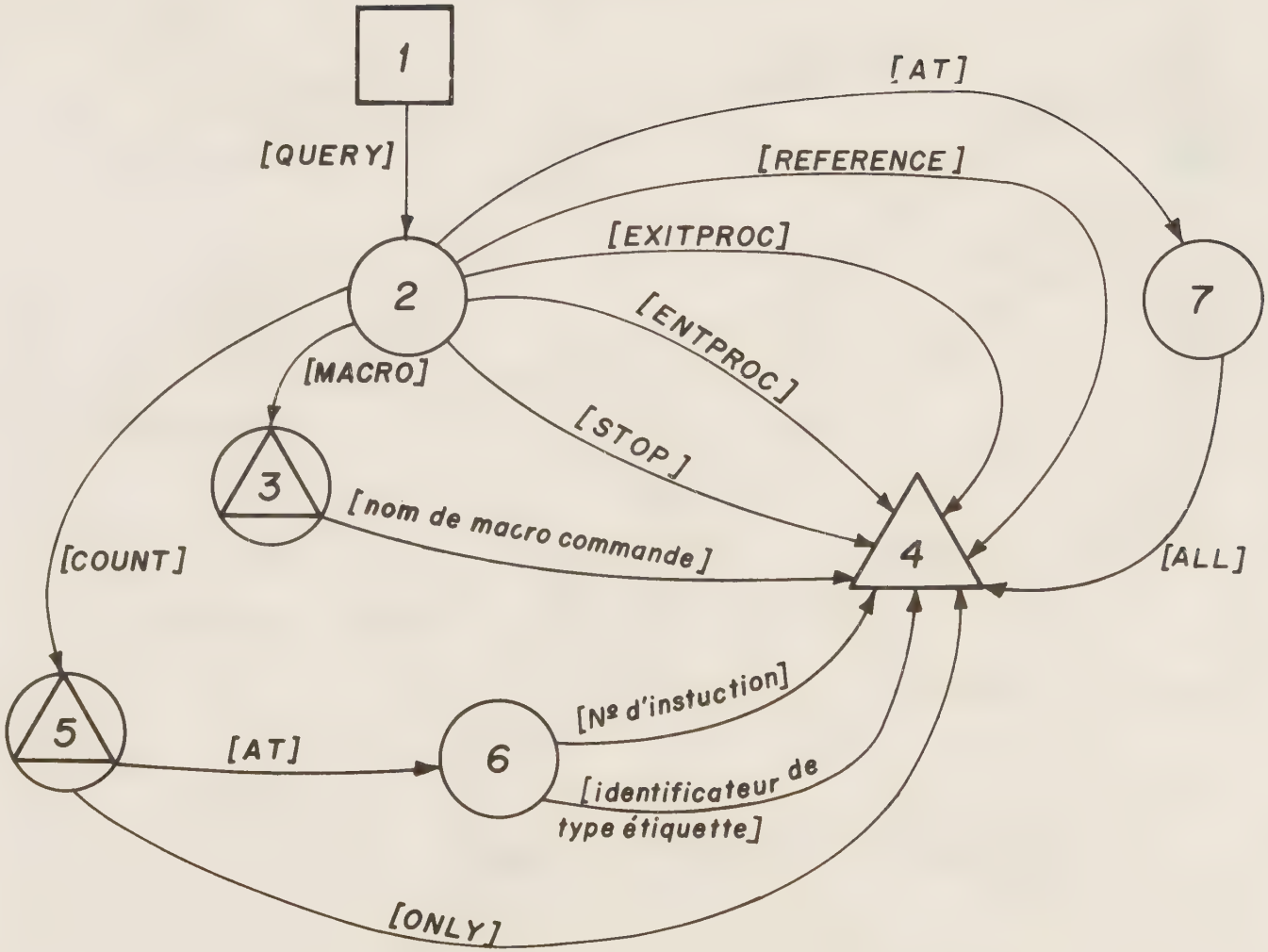
Commande d'affichage et
de modification



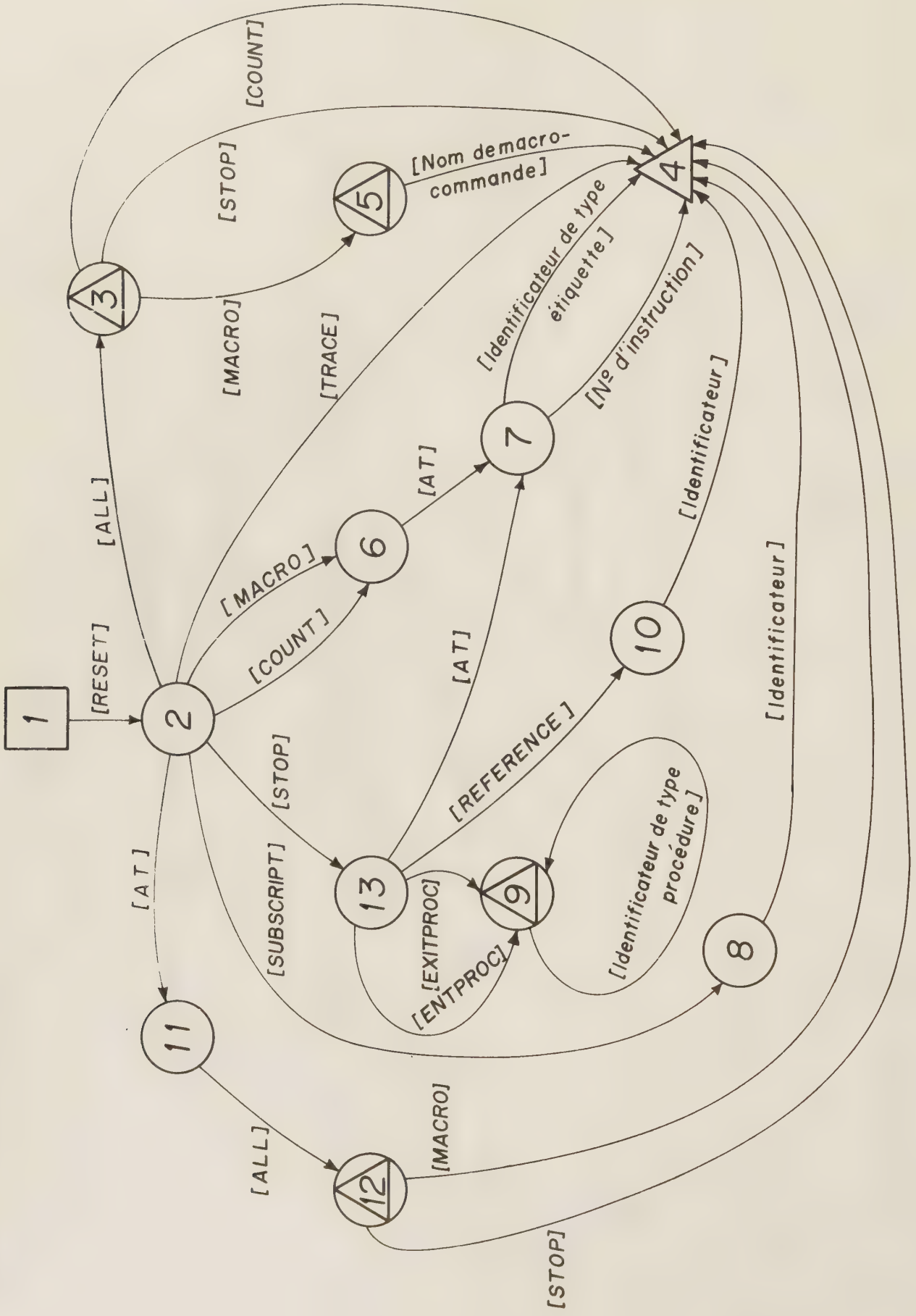
Commande de contrôle de l'exécution

* Remarque : Si le nom de la macro n'exi[illegible] pas, le système se met en attente [illegible] ture des lignes composant la macro-[illegible] La rencontre de la fin de macro-comma[illegible] (ENDMAC) provoque la fin de la lecture.




1
[QUERY]
2
[AT]
[REFERENCE]
[EXITPROC]
[ENTPROC]
[STOP]
7
[MACRO]
3
[nom de macro commande]
4
[ALL]
[COUNT]
5
[AT]
6
[Nº d'instuction]
[identificateur de type étiquette]
[ONLY]

Commande d'annulation RESET



BIBLIOGRAPHIE

[A1] AKIYAMA Fumio

An example of Software system debugging.
AFIP Congress 71 Ljubljana August 1971

[B1] BAER J.L and CAUCHEY R.

Segmentation and optimization of programs from structures analysis.
Spring Joint Computer Conference, 1972.

[B2] BALZER R.M

EXDAMS. Extendable Debugging and Monitoring System.
Spring Joint Computer Conference, 1969.

[B3] BAYER R.

The Alcor illinois 7090/7094 post moxtem dump.
Communication of the ACM Volume 10/Number 12/December, 1967.

[B4] BERTHAUD M.

Etude d'un compilateur incrémental pour le compilateur PL/1.
Centre Scientifique IBM de Grenoble, Etude FF2-0097.

[B5] BERTHAUD M., CLAUZEL D., JACOLIN M.

GSL (Grenoble System Language) Définition du Langage.
Centre Scientifique IBM de Grenoble, Etude FF2-0133.

[B6] BERTHAUD M., GRIFFITHS M.

Incremental Compilation Conversational Interpretation.
Annual review in automatic programming, 1973.

[B7] BERTHAUD M., JACOLIN M., POTIN Ph., SAVARY H.

Coupling Virtual Machine and System Construction.
ACM Sigarch-Sigops Workshop on Virtual Computer Systems, 1973.

[B8] BERTHAUD M. (Mme), SAVARY H.

Etude et Réalisation d'un Macro Langage Conversationnel.
Rapport de stage de D.E.A., Juin 1971.

[B9] BOBRAW D. G.

Tenex a paged time sharing system for PDP10.
Communication of the ACM Volume 15/Number 3/March, 1972.

[C1] CLAUZEL D.

Register management for code optimization in one-pass compiler.
A paraître.

[C2] COMON D.

Gestion de bibliothèque de programmes dans un grand projet.
AFCET Congrès de l'AFCET, 1972 Grenoble.

[C3] CONROW K. and SMITH R.G.

NEATER2 : A PL/1 source statment reformatter.
Communication of the ACM Volume 13/Number 11/November, 1970.

[C4] CONWAY R.W. and WILCOX T.R.

Design and implementation of a diagnostic compiler for PL/1.
Communication of the ACM Volume 16/Number 3/March, 1973.

[C5] CURRIE I.F.

Secure Library Systems.
Presented at the Advanced Course on Computer Systems Architecture.
4-15 Decembre 1972, L'Alpe d'Huez France.

[D1] DAHL O.J., DIJKSTRA E.W. and HOARE C.A.R.

Structured programming.
Academic press London and New-Yord, 1972.

[D2] DANIERI J.J.

Utilisation du système de test et d'évaluation de programmes (STEP) pour la mise au point de programmes.
AFCET Congrès de l'AFCET, 1972 Grenoble.

[D3] DEVILLERS C. et SAVARY H.

Chargeur d'IPL assisté.
Note interne à paraître.

[D4] DIJKSTRA E.W.

The Structure of the "THE" Multiprogramming System.
Communication of the ACM Volume 11/Numver 3/May, 1968.

[D5] DIJKSTRA E.W.

Structured programming.
Report on a conference Spensered by the NATO Science Committec Software Engineering Technique, Rome April 1970.

[E1] ELMENDORF W.R.

"About testing"
IEEE Transactions on Systems Sciences and Cybernetics
Vol. SSC-5 . N°4 October 1969.

[E2] ELSPAS B., LEVITT K.N., WALDINGER R.J., WAKSMAN A.

An assessment of techniques for proving program correctness.
ACM Computing Survey Volume 4/Number 2/June, 1972.

[E3] EVANS T.G. and DARLEY L.D.

DEBUG - An extension to current online debugging techniques
Communication of the ACM Volume 8/Number 5/May, 1965.

[F1] FERGUSON H.E. and BERNER E.

Debugging systems at the source language level.
Communication of the ACM Volume 6/Number 8/August, 1963.

[F2] FLOYD R.W.

Toward interactive design of correct programs.
Standford artificial intelligence project.
Memo no AIM-50 Computer science departments
Report N° CS-235.

[G1] GRAHAM M.R., CLANEY G.J., DE VANEY D.B.

A Software Design and Evaluation System.
Communication of the ACM Volume 16/Number 2/February, 1973.

[G2] GRIFFITHS M.

Analyse Déterministe et Compilateurs
Faculté des Sciences de Grenoble Thèse d'Etat, 1969.

[G3] GRIFFITHS M., PECCOUD M., PELTIER M.

Incremental Interactive Compilation
Proc I.F.I.P, August, 1968.

[G4] GRISHMAN R.

Criteria for a debugging language.
Debugging techniques in Large Systems
Courant Computer Science Symposium 1. 1969.
Edited by Randall Rustin.

[G5] GOETZ M.A.

The Monopolized Software Industry.
Datamation, October, 1972.

[H1] HANS C., LEFEBVRE P., PETEUL-HARMEL B.

Mise au point conversationnelle de système grâce à un couplage particulier de machines virtuelles.
Canadian Computer Conference, Session 1972.

[H2] HOPCROFT J.E. and ULLMAN J.D.

Formal Languages and their relation to automata.
Addison-Wesley Publishing-company.

[I1] IBM Control Program 67/Cambridge Monitor System VL3.0
User's guide manual n° GH 20-0859.

[I2] IBM System/360 operating system.
TESTRAN C28-6648-1.

[I3] IBM System/360 Time Sharing System,
Time Sharing Support System Form C28-2006-0.

[I4] IBM System/360 Operating System.
PL/1 (F) Language Reference Manual.

[I5] IBM System/360 Operating System.
FORTRAN IV Language Form C28-6515.7.

[I6] IBM 7090/7094 IBSYS Operating System V3.
IBJOB Processor Debugging Pachage Form C28-6393-2.

[I7] IBM System/360 Operating System Time Sharing Option
Commande Language Reference Order N° GC28-6732-3.

[I8] IBM Control-program 67.
Program Logical Manual - manual N° GY20-0590.

[I9] IBM System/370.
Principle of operation GA 22-7000-2

[I10] IBM System/360 Operating System.
Loader Program Logic Manual GY28-6714-1.

[J1] JACOLIN M.

Mise au point interactive des programmes en langage évolué.
Canadian Computer Conference, Session 1972.

[J2] JACOLIN M.

Mise au point conversationnelle de programmes Fortran : FORTBUG
Centre scientifique IBM de Grenoble Etude N°FF2.0136
Juin 1972.

[K1] KING J.

A verifying Compiler.
Debugging techniques in larges systems.
Courant Computer Science Symposium 1, 1969.
Edited by Randall Rustin.

[K2] KULSRUD H.E.

"HELPER" : interactive extensible debugging system.
Second Symposium on Operating Systems Principle 1969.

[L1] DE LAMBERTERI X.

Espace virtuel et gestion de fichier.
Université Scientifique et Médicale de Grenoble,
Thèse de 3ème cycle., Juin 1973.

[L2] LANG C.A.

Languages for writing systems programs.
Report on a conference spendered by the NATO science comittee.
Software engeneering techniques Rome, April 1970.

[L3] LEFEBVRE P.

SPY : Un système de contrôle.
La mise au point de système de programmation grâce à un couplage de machines virtuelles.
Université scientifique et médicale de Grenoble,
Thèse de 3ème cycle, Juin 1972.

[L4] LEPEUVE Martine

Application des graphes à l'optimisation d'un modèle destiné à mesurer les performances d'un système d'exploitation.
Université scientifique et médicale de Grenoble,
Thèse, Janvier 1973.

[M1] MILLS HARLEN D.

Top down programming in large systems.
Debugging Computer Science Symposium 1, 1969.
Edited by Randall Rustin.

[M2] MILLS H.D.

The impact of structured programming on software engineering and production.
Professional Development Seminar, 1/2 February 1973, Paris
Chapitre Français de l'ACM.

[M3] MOULTON P.G and MULLER M.F.

DITRAN - A Compiler Emphasizing Diagnostics.
Communication of the ACM Volume 10/Number 1/January, 1967.

[M4] M.T.S. The Michigan Terminal System
Volume 1 : MTS and the Computing Center.
University of Michigan Computing Center Ann Arbor, Michigan.

[N1] NAUR P.

Revised Report on the Algorithmic Language ALGOL 60.
Regnecentralen, Copenhagen, 1962 and elsewhere.

[N2] NGUYEN-THANH THI

XCP Un environnement graphique conversationnel pour l'examen d'un système.
Application à CP67
Université scientifique et médicale de Grenoble,
Thèse de 3ème cycle., Juin 1973.

[O1] ORGANICK Elliot I.

The multies system : An examination of its structures.
The MIT press Cambridge Massachussetts and London England.

[P1] PARNAS D.L.

On the criteria to be used in decomposing systems into modules.
Communication of the ACM Volume 15/Number 12/December, 1972.

[P2] PARNAS D.L.

A technique for software module specification with examples.
Communication of the ACM Volume 15/Number 5/May, 1972.

[P3] PETEUL-HARMEL B.

La mise au point de programmes par simulation. Réalisation d'un support conversationnel de mise au point : PILOTE.
Université scientifique et médicale de Grenoble,
Thèse de 3ème cycle, Juin 1971.

[P4] POMEROY J.W.

A guide to programming tools and techniques.
IBM Systems journal Volume eleven /Number three/1972.

[P5] POOL Peter

Bebugging and testing.
Advanced course on Software Engineering.
Lectures notes in economics and mathematical systems. 81.
Springer-Verlag, 1973.

[P6] PRESSER L., WHITE J.

Linkers and Loaders.
ACM Computing Survey Volume 4/Number 3/September, 1972.

[S1] VACKMAN H., ERICKSON W.J., GRANT E.E.

Exploratory experimental studies comparing on line and offlin
programming performance.
Communication of the ACM Volume 11/Number 1/January, 1968.

[S2] SCHROEDER H.D. and SALTSER J.H.

A hardware architecture for implementing protection rings.
Communication of the ACM Volume 15/Number 3/March, 1972.

[S3] SPILLMAN T.C.

Explosing side-effects in PL/1 optimizing compiler.
I.F.I.P. Congress 71 Ljubljana, August 1971.

[S4] VER STEEG R.L.

Talk - A high level source language debugging technique with
real time data extraction.
Communication of the ACM Volume 7/Number 7/July, 1967.

[S5] STOCKON GAINES R.

"The debugging of computer programs"
August 1969. Faculty of Princeton University.

[T1] TISSERAND et GUIARD

O.D.A.S.S. Outil de documentation automatique des symboles d'un système.
A.F.C.E.T. Congrès A.F.C.E.T., 1972 Grenoble.

[T2] TSICHRITZIS D.

Project management.
Advanced course on software engineering.
Lecture notes in economics and mathematical systems. 81.
Springer-Verlag, 1973.

[T3] TSICHRITZIS D.

Reliability.
Advanced course on software engineering.
Lecture notes in economics and mathematical systems. 81.
Springer-Verlag, 1973.

[U1] UNIVAC - Time sharing UNIVAC 1108
Fortran Conversationnel.

[W1] VAN WIJNGAARDEN A., MAILLOUX B.J., PECK J.E.L., KOSTER C.H.A.

Report on the algorithmic language ALGOL 68.
Second printing, Mathematisch centrum, Amsterdam.
MR 101, octobre 1969.

[W2] WIRTH N. and HOARE C.A.R.

A contribution to the development of Algol.
Communication of the ACM Volume 9/Number 6/June, 1966.

[W3] WIRTH N.

The programming language PASCAL.